

Windows

操作系统原理

尤晋元 史美林

陈向群 向勇 王雷 编著
郑扣根 陈英 马洪兵



机械工业出版社
China Machine Press

TP316.7

20

重点大学计算机教材

Windows操作系统原理

尤晋元 史美林
陈向群 向 勇
王 雷 郑扣根
陈 英 马洪兵

编著



机械工业出版社
China Machine Press

Congratulatory letter

Bill Gates



PCs have revolutionized the way we live, work, learn and play, and transformed how we communicate with each other. They have empowered people to be more creative and businesses more efficient, opened up amazing opportunities for education, and helped stimulate productivity worldwide. Combined with inexpensive, widespread Internet access, they've helped news and information travel faster and more freely than ever before, and broken down borders between nations, people and economies. So it's no surprise that more than half of PC users now consider it the most important device in their home.

Operating System (OS) is the brain and soul of a computer. From DoS to Windows to .NET, our platform have indeed transformed the way people use PC with enabling technologies that make PC a centerpiece of productivity, communications, and entertainment for home and enterprise as well.

I am very pleased to see the birth of first-ever Chinese textbook on Windows OS, with the tremendous efforts from top Chinese academia, support from Chinese government, and collaboration with Microsoft Research in China.

This year marks the 20th anniversary of PCs. In the coming decade, the PC will continue to become more powerful, less expensive and even more essential - probably the most important tool people use to work, play and stay in touch. If the past 20 years have been impressive, the next 20 will be astounding.

William H. Gates
Chairman and Chief Software Architect
Microsoft Corporation
August 2001

JS820/OP

本书是在微软中国研究院和美国微软公司的支持下，由美国微软公司提供Windows全面内部技术资料，全国五所知名重点大学操作系统主讲教师组成写作组，历时一年写作完成的一本以Windows 2000/XP为实际示例，讲授计算机操作系统原理的教科书。

本书讲述了当代计算机操作系统的原理，并具体分析了操作系统原理在Windows 2000/XP中的实现技术和方法，有理论、有示例。既有设计思想精要的提炼，又有具体实现细节的分析。

本书是第一本用中文出版的、讨论Windows操作系统原理的教科书，也是第一本将讲授操作系统一般原理与分析Windows操作系统内部体系结构相结合的书籍。本书适合作为高等院校计算机和电子工程相关专业的操作系统教科书，也是一本用于设计、开发基于Windows的应用软件，以及编写Windows操作系统驱动程序的重要参考书。同时本书也是微软Windows 2000/XP平台上应用软件设计和开发人员的必备参考书。

版权所有，侵权必究。

本书所涉及的部分技术内容已获得微软出版社的授权。

图书在版编目（CIP）数据

Windows操作系统原理 / 尤晋元等编著. —北京：机械工业出版社，2001.8

（重点大学计算机教材）

ISBN 7-111-09211-2

I. W… II. 尤… III. 窗口软件，Windows—高等学校—教材 IV. TP316.7

中国版本图书馆CIP数据核字（2001）第051431号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：姚 蕾

北京牛山世兴印刷厂印刷·新华书店北京发行所发行

2001年8月第1版第1次印刷

787mm × 1092mm 1/16 · 29印张

印数：0 001-10 000册

定价：39.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

Congratulatory letter

Bill Gates



PCs have revolutionized the way we live, work, learn and play, and transformed how we communicate with each other. They have empowered people to be more creative and businesses more efficient, opened up amazing opportunities for education, and helped stimulate productivity worldwide. Combined with inexpensive, widespread Internet access, they've helped news and information travel faster and more freely than ever before, and broken down borders between nations, people and economies. So it's no surprise that more than half of PC users now consider it the most important device in their home.

Operating System (OS) is the brain and soul of a computer. From DoS to Windows to .NET, our platform have indeed transformed the way people use PC with enabling technologies that make PC a centerpiece of productivity, communications, and entertainment for home and enterprise as well.

I am very pleased to see the birth of first-ever Chinese textbook on Windows OS, with the tremendous efforts from top Chinese academia, support from Chinese government, and collaboration with Microsoft Research in China.

This year marks the 20th anniversary of PCs. In the coming decade, the PC will continue to become more powerful, less expensive and even more essential - probably the most important tool people use to work, play and stay in touch. If the past 20 years have been impressive, the next 20 will be astounding.

William H. Gates
Chairman and Chief Software Architect
Microsoft Corporation
August 2001

JS820/op

Congratulatory letter

Bill Gates



PCs have revolutionized the way we live, work, learn and play, and transformed how we communicate with each other. They have empowered people to be more creative and businesses more efficient, opened up amazing opportunities for education, and helped stimulate productivity worldwide. Combined with inexpensive, widespread Internet access, they've helped news and information travel faster and more freely than ever before, and broken down borders between nations, people and economies. So it's no surprise that more than half of PC users now consider it the most important device in their home.

Operating System (OS) is the brain and soul of a computer. From DoS to Windows to .NET, our platform have indeed transformed the way people use PC with enabling technologies that make PC a centerpiece of productivity, communications, and entertainment for home and enterprise as well.

I am very pleased to see the birth of first-ever Chinese textbook on Windows OS, with the tremendous efforts from top Chinese academia, support from Chinese government, and collaboration with Microsoft Research in China.

This year marks the 20th anniversary of PCs. In the coming decade, the PC will continue to become more powerful, less expensive and even more essential - probably the most important tool people use to work, play and stay in touch. If the past 20 years have been impressive, the next 20 will be astounding.

William H. Gates
Chairman and Chief Software Architect
Microsoft Corporation
August 2001

JS820/OP

序（一）

杨芙清

进入新世纪以来，中国的高技术产业，特别是信息产业持续快速发展，信息技术已经成为21世纪经济发展的驱动力。

信息是客观事物状态和运动特征的一种普遍形式。人类抽象的经验和知识正逐步由软件予以精确地体现。而软件是人类知识的固化，是知识经济的基本表征，它已成为信息时代的新型“物理设施”。



软件是信息化的核心。国民经济和国防建设、社会发展、人民生活都离不开软件，软件无处不在。软件产业是增长最快的朝阳产业，是具有高附加值、高投入/高产出、无污染、低能耗的绿色产业。软件产业关系到国家经济安全和文化安全，体现了国家综合实力，是决定21世纪国际竞争地位的战略性产业。

计算机操作系统正是软件技术含量大、附加值高的部分，是软件系统的核心，是软件基础运行平台的主要成份。

在操作系统的商业产品市场上，微软公司在20世纪80年代初期为IBM公司的个人计算机配置了PC-DOS操作系统，几年之后，又推出了Windows操作系统。由于Windows采用了图形界面，易学易用，又辅之良好的市场策略，因此它逐渐占据了个人计算机市场的主要地位，并成为主要的操作系统产品。

本书是在美国微软公司提供Windows操作系统内部技术资料的基础上，讲述计算机操作系统原理的教科书。参加本书写作的作者均是国内一些重点高等

贺词

个人电脑革命性地改变了人们生活、学习和娱乐的方式,同时它也极大地促进了人们沟通的方式。它的出现,激发了人类的创造性,有力地促进了商业效率的提高;它的出现,还为教育事业创造了前所未有的机遇,使整个世界的生产力向前迈进了一大步。结合廉价、广泛的互联网接入,个人电脑更为信息提供了空前高效、开放的传播途径,并且打破了国家、种族和经济上的壁垒。由此,半数以上的个人电脑用户把它看作是家庭中最重要设备也就不足为奇了。

操作系统是电脑的核心和灵魂。从DOS发展到Windows 继而是.NET,我们的平台借助科学技术从本质上改变了人们使用个人电脑的方式,使它成为搭建生产力、通讯、家庭和企业娱乐桥梁的核心枢纽。

我非常欣慰地看到第一个 Windows 操作系统中文教科书的诞生,这离不开来自中国权威学术界的努力、中国政府的鼎力支持和微软中国研究院的通力合作。

2001年标志着个人电脑诞生了20周年。在未来的十年中,个人电脑将持续发展,变得更具影响力也更便宜,它将被更为广泛地应用于工作、娱乐和日常联络,成为生活中不可或缺的、最重要的工具。如果把过去20年的成长历程称之为令人难忘的话,那么未来20年的发展将则更是令人期待的。

比尔·盖茨
微软主席兼首席软件设计师
2001年8月

序（一）

杨芙清

进入新世纪以来，中国的高技术产业，特别是信息产业持续快速发展，信息技术已经成为21世纪经济发展的驱动力。

信息是客观事物状态和运动特征的一种普遍形式。人类抽象的经验和知识正逐步由软件予以精确地体现。而软件是人类知识的固化，是知识经济的基本表征，它已成为信息时代的新型“物理设施”。



软件是信息化的核心。国民经济和国防建设、社会发展、人民生活都离不开软件，软件无处不在。软件产业是增长最快的朝阳产业，是具有高附加值、高投入/高产出、无污染、低能耗的绿色产业。软件产业关系到国家经济安全和文化安全，体现了国家综合实力，是决定21世纪国际竞争地位的战略性产业。

计算机操作系统正是软件技术含量大、附加值高的部分，是软件系统的核心，是软件基础运行平台的主要成份。

在操作系统的商业产品市场上，微软公司在20世纪80年代初期为IBM公司的个人计算机配置了PC-DOS操作系统，几年之后，又推出了Windows操作系统。由于Windows采用了图形界面，易学易用，又辅之良好的市场策略，因此它逐渐占据了个人计算机市场的主要地位，并成为主要的操作系统产品。

本书是在美国微软公司提供Windows操作系统内部技术资料的基础上，讲述计算机操作系统原理的教科书。参加本书写作的作者均是国内一些重点高等

序（一）

杨芙清

进入新世纪以来，中国的高技术产业，特别是信息产业持续快速发展，信息技术已经成为21世纪经济发展的驱动力。

信息是客观事物状态和运动特征的一种普遍形式。人类抽象的经验和知识正逐步由软件予以精确地体现。而软件是人类知识的固化，是知识经济的基本表征，它已成为信息时代的新型“物理设施”。



软件是信息化的核心。国民经济和国防建设、社会发展、人民生活都离不开软件，软件无处不在。软件产业是增长最快的朝阳产业，是具有高附加值、高投入/高产出、无污染、低能耗的绿色产业。软件产业关系到国家经济安全和文化安全，体现了国家综合实力，是决定21世纪国际竞争地位的战略性产业。

计算机操作系统正是软件技术含量大、附加值高的部分，是软件系统的核心，是软件基础运行平台的主要成份。

在操作系统的商业产品市场上，微软公司在20世纪80年代初期为IBM公司的个人计算机配置了PC-DOS操作系统，几年之后，又推出了Windows操作系统。由于Windows采用了图形界面，易学易用，又辅之良好的市场策略，因此它逐渐占据了个人计算机市场的主要地位，并成为主要的操作系统产品。

本书是在美国微软公司提供Windows操作系统内部技术资料的基础上，讲述计算机操作系统原理的教科书。参加本书写作的作者均是国内一些重点高等

序（二）

张亚勤

作为计算机技术的重要基础学科，操作系统近年来在概念和技术上都有很快的发展。因此，一本好的操作系统教材，除了要有系统的基础理论介绍之外，还应该结合当前主流操作系统的实例，这样才能反应最新的技术动态，使学生们学有所用。



在2000年夏季由微软公司主办的第一届中国高校计算机系主任座谈会上，不少老师谈到：在中国，Windows 操作系统已经被广泛地应用于政府、教育和商业等各个领域，而一个普遍存在的问题是：用户对Windows的机制和内核技术了解得不够，从而在很大程度上影响了人们更好地使用Windows操作系统。

诚然，作为一个现代操作系统，Windows不断地推陈出新。随着Windows NT、Win CE、Windows 2000和不久即将面世的Windows XP及.NET的不断推出，Windows不再是传统意义上的桌面计算的操作系统，它正在成为一个网络环境下的平台。

从这个意义上来讲，Windows作为操作系统教学的实例，既能反映当前操作系统技术的发展方向，又能结合当前的应用实际，是一个非常理想的操作系统教学平台。

目前，国内涉及Windows的使用、维护和开发的书很多。但作为教材，系统讲解Windows的机制和内核，并适合于中国高校教学特点的书还没有。要想写

院校从事计算机操作系统课程教学和科研的老师。这些老师把在操作系统课程上的长期教学经验、科研积累同分析Windows操作系统内部技术原理结合起来，写作完成了这本操作系统教科书。

我相信，本书的出版会对使用Windows操作系统的读者有所帮助。

中国科学院院士 北京大学教授

杨芙清

2001年7月

序（二）

张亚勤

作为计算机技术的重要基础学科，操作系统近年来在概念和技术上都有很快的发展。因此，一本好的操作系统教材，除了要有系统的基础理论介绍之外，还应该结合当前主流操作系统的实例，这样才能反应最新的技术动态，使学生们学有所用。



在2000年夏季由微软公司主办的第一届中国高校计算机系主任座谈会上，不少老师谈到：在中国，Windows 操作系统已经被广泛地应用于政府、教育和商业等各个领域，而一个普遍存在的问题是：用户对Windows的机制和内核技术了解得不够，从而在很大程度上影响了人们更好地使用Windows操作系统。

诚然，作为一个现代操作系统，Windows不断地推陈出新。随着Windows NT、Win CE、Windows 2000和不久即将面世的Windows XP及.NET的不断推出，Windows不再是传统意义上的桌面计算的操作系统，它正在成为一个网络环境下的平台。

从这个意义上来讲，Windows作为操作系统教学的实例，既能反映当前操作系统技术的发展方向，又能结合当前的应用实际，是一个非常理想的操作系统教学平台。

目前，国内涉及Windows的使用、维护和开发的书很多。但作为教材，系统讲解Windows的机制和内核，并适合于中国高校教学特点的书还没有。要想写

序（二）

张亚勤

作为计算机技术的重要基础学科，操作系统近年来在概念和技术上都有很快的发展。因此，一本好的操作系统教材，除了要有系统的基础理论介绍之外，还应该结合当前主流操作系统的实例，这样才能反应最新的技术动态，使学生们学有所用。



在2000年夏季由微软公司主办的第一届中国高校计算机系主任座谈会上，不少老师谈到：在中国，Windows 操作系统已经被广泛地应用于政府、教育和商业等各个领域，而一个普遍存在的问题是：用户对Windows的机制和内核技术了解得不够，从而在很大程度上影响了人们更好地使用Windows操作系统。

诚然，作为一个现代操作系统，Windows不断地推陈出新。随着Windows NT、Win CE、Windows 2000和不久即将面世的Windows XP及.NET的不断推出，Windows不再是传统意义上的桌面计算的操作系统，它正在成为一个网络环境下的平台。

从这个意义上来讲，Windows作为操作系统教学的实例，既能反映当前操作系统技术的发展方向，又能结合当前的应用实际，是一个非常理想的操作系统教学平台。

目前，国内涉及Windows的使用、维护和开发的书很多。但作为教材，系统讲解Windows的机制和内核，并适合于中国高校教学特点的书还没有。要想写

成这样一本教材，是离不开中国高校操作系统教师的努力和微软公司的支持的。

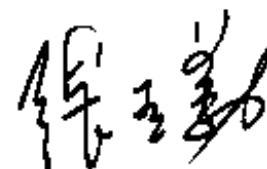
微软中国研究院是微软公司在亚太地区唯一的基础研究机构。成立三年以来，除了从事世界一流的基础研究之外，我们一直把支持中国高校的课程和教材建设、帮助中国培养高水平的计算机人才作为非常重要的工作。

我欣喜地看到，8位来自中国著名高校的操作系统专家不断地努力，从去年的全国操作系统课程研讨会上的倡议，到今年去微软公司了解Windows源代码，再到与微软公司Windows组的设计师共同探讨。在短短的一年多的时间内，终于把编写一本中文Windows操作系统教材的设想变成了现实。

浏览全书，我感到这本书既包含了对操作系统原理的系统讲解，又能紧密结合Windows的内核技术，其中还配有大量相关的实例、试验和作业，确实是一本目前难得的通过Windows实例讲解现代操作系统的教材。

最后，我想借此机会代表比尔·盖茨（Bill Gates）先生及微软公司表示对支持中国教育事业的长期承诺和诚意。我们希望今后能和中国高校一起，交流计算机技术的最新动态，写出更多这样的好教材，为中国信息产业的腾飞尽一份力。

微软中国研究院院长



2001年8月

编者的话

《Windows 操作系统原理》正式出版了，这是一件值得高兴的事。这本书可能是世界上第一本用中文出版的、讨论Windows 操作系统原理的教科书；这本书也是分析Windows 操作系统内部体系结构的少数几本书籍之一。

我们希望本书可用作中国高等院校相关专业操作系统课程的教材或权威参考书，其读者对象应该是高等院校相关专业的教师、研究生、本科生。我们还希望本书也是微软Windows 2000/XP平台上应用软件设计和开发人员的必备设计参考书，那么，另一读者群应该是国内在微软Windows 2000/XP平台上从事设计和开发工作的应用软件设计开发人员。当然，这本书的真正价值，需要由本书的读者去评价。不过，回顾这本书的策划、写作、编辑和出版的过程，其中确实有不少故事，值得我们去回忆。

进入新世纪以来，在中国的信息产业界，上至高级主管部门，下至普通计算机用户，都对计算机操作系统投入了前所未有的关心和注视。人们普遍认识到，计算机操作系统是整个信息技术领域中一块极其重要的基石。要构建现代化的、稳固而又可靠的信息技术大厦，不掌握计算机操作系统是行不通的。

2000年首届由微软公司主办的高等院校计算机系主任座谈会上，很多系主任提出希望能有一本更多讲解Windows操作系统原理的教材。2000年10月在清华大学举行的第六届全国操作系统课程教学研讨会上，许多主讲操作系统课程的教师也提出希望更多地了解Windows内核原理，以供教学需要。谁都清楚，当代使用最广泛、最普及的计算机操作系统是Windows 操作系统。而另一个事实是，到本书出版之前，还没有哪一家国内外的出版社出版过一部详细介绍和

分析Windows 操作系统内部原理的教材。

正是在这样的大环境下，写作一本结合当代Windows操作系统、讲授操作系统原理的教科书的构想成熟了。

思想的火花在微软中国研究院的支持下，很快燃成了一支火炬。北京大学计算机科学技术系的陈向群老师承担了组织写作班子的工作。清华大学计算机系的史美林老师和向勇老师、清华大学电子工程系的马洪兵老师、北京航空航天大学计算机系的王雷老师、浙江大学计算机系的郑扣根老师、上海交通大学计算机系的尤晋元老师和陈英老师，都欣然接受了邀请。他们都是各自学校的操作系统课程的主讲教师，曾写作、编辑和翻译出版过多本有关操作系统原理的教科书，而且目前都在主持有关操作系统的科研工作。毫无疑问，有了这些老师们组成的写作组，本书的写作质量就有了保证。

详细的写作提纲拟就出来后，问题也接踵而来。要分析、介绍Windows操作系统原理，不能不了解Windows 操作系统的内部结构，也不能不去阅读Windows操作系统的源代码。但是，在国内外已经出版的计算机书籍和资料中，包括微软出版社出版的书籍，没有一本书能够提供这两个方面的具体资料。

怎么办？问题很快通过张高博士反映到了微软中国研究院高校关系与运作部负责人陈宏刚博士那里，也反映到美国微软公司总部。经过微软中国研究院的努力，在Windows开发组工程师Dave Probert的大力支持下，美国微软公司总部作出决定，允许写作组的老师们阅读Windows 操作系统的源代码，并提供有关Windows 操作系统的内部结构的参考资料。这一消息让我们感到非常高兴。

可是，良好的愿望有时并不能成为现实。由于中国和美国在有关法律上的差异，美国微软公司尚不能在中国本土向老师们提供阅读Windows操作系统源代码的条件。怎么办？

接着发生的事情是谁也没有想到的。

美国微软公司总部决定，由微软中国研究院出资，邀请全体写作组的成员到美国微软公司总部西雅图阅读Windows 操作系统的源代码，并由主持

VIII

Windows 2000操作系统设计的高级技术人员，向中国写作组专门讲解Windows操作系统的内部体系结构。据微软中国研究院介绍，这是美国微软公司第一次在亚洲地区开放Windows操作系统的源代码。

对于美国微软公司给予本书写作上的全力支持，我们表示真诚的感谢！

四月的西雅图，风光秀丽，景色宜人。我们一行四名老师和微软中国研究院的陈宏刚博士、王瑾小姐非常幸运地赶上了郁金香节，成片成片的郁金香，在绿树和草地的环绕下，显得格外艳丽。美国微软公司总部坐落在西雅图的雷德蒙镇上。公司总部的办公楼，一半被浓密的森林包围着，另一半则是波光粼粼的湖泊。不时有一艘汽艇飞快地划过平静的湖面，浪花串起一条条闪光的银链。远处湖对岸的树林中，时隐时现的是一座座别墅小园。微软公司总部的不少高级主管和软件工程师们的家庭就住在这些建筑造型别致、风格各异的湖畔别墅群中。我们在美国期间，许多事情令人难忘，例如，在一次出行中，由于不熟悉路线，违反了交通规则，然而，当警察知道我们是从遥远的中国来的时候，没有罚款，并且耐心地指明路线方向。

写作组在西雅图的工作是紧张和短暂的。每天一半的时间是到机器上阅读Windows操作系统的源代码，另一半时间是听取美国微软公司总部Windows2000设计师Dave Probert组织的讲课。晚上则是开会讨论问题或者整理笔记和技术资料。

为了能够更深入地了解内部技术细节，上海交通大学的尤晋元老师在成行前就安排了一些研究生专门列出有关Windows的内部技术问题，以便利用这次宝贵的机会和美国微软公司总部的高级技术人员进行探讨。

由于工作安排上的原因，整个写作组的八名成员，有四位未能去西雅图访问。但是他们也随时和赴美访问的老师保持着热线联系；史美林老师当时正在欧洲访问，也不断通过电话和电子邮件关心写作小组工作的进展情况。

专门给写作组讲课的Dave Probert，曾经是美国Unix技术公司的主要研发人员，从事Unix操作系统的设计和开发长达二十年，对Unix系统有着极其深刻的

认识；目前，他是Windows内核开发组骨干成员。像Dave Probert这样不可多得的高级软件技术人才，在美国微软公司数以百计。他们中有许多曾经从事过Unix操作系统、DEC VAX 机操作系统、Sun 工作站操作系统等等国际知名操作系统研究和设计的关键技术人员。在世界上很难找到第二家公司，像美国微软公司这样，拥有一大批经验丰富的操作系统研究和设计人员。

在西雅图工作期间，写作组会见了许多在美国微软公司总部工作的人员，其中有美国人、中国人，也有其他国家的人员。不论他们从事的是哪一种工作，每一名工作人员都表现出一种极为认真的敬业精神。写作组所到之处，都能感觉到美国微软公司人员的一种发自内心的、热诚合作的态度。写作组把周六和周日都用来加班，微软中国研究院和微软公司总部的有关软件技术人员也都一起陪同，毫无怨言。

写作组也见到了不少微软公司的程序设计人员，他们的年龄在40余岁，甚至50余岁，仍然在从事编程工作，并且乐此不疲。这与中国国内一些IT公司对于才过了30岁的软件设计应聘人员都不想使用的现象，实在是大相径庭。

毫无疑问，本书写作组在美国微软公司总部的有关工作，对本书的顺利成稿起到了关键的作用。

今年5月和6月，写作小组又两次在微软中国研究院集中，共同讨论修改稿子，直到6月底，最终完成书稿。

本书的读者对象主要是在高等院校学完了“操作系统原理”课程，需要深入了解Windows操作系统的学生；在各类机构工作，需要了解Windows 操作系统的内部体系结构，以便更好地在Windows之上设计开发各类应用软件的技术人员；需要了解如何编写Windows 操作系统驱动程序和应用程序的软件工程师。当然，本着开卷有益的宗旨，凡是希望了解Windows 操作系统内部原理的人们，都会从本书中得到启示。

《Windows操作系统原理》一书的结构和编排布局如下：全书共分为9章，第1章操作系统概述（陈向群）；第2章Windows 2000/XP体系结构（陈向



(部分编写组成员，2001年6月下旬摄于微软中国研究院)

第一排：陈向群、王瑾

第二排：马洪兵、陈英、王雷、郑扣根

第三排：马歆、刘佐扬、陈宏刚、向勇



左起：郑扣根、陈向群、David Cutler、尤晋元、王雷

(2001年4月下旬摄于微软总部David Cutler办公室前，David Cutler是Windows NT的总设计师，是全微软公司十几位Distinguished Engineer之一，并且是唯一的一位Senior Distinguished Engineer。)

群)；第3章进程及处理器管理(向勇、伍尚广、史美林)；第4章存储管理(王雷)；第5章文件系统(郑扣根)；第6章I/O系统(陈向群)；第7章网络(陈英、尤晋元)；第8章Windows应用程序设计(马洪兵)；第9章Windows设备驱动程序设计(马洪兵)。

本书在出版过程中得到了美国微软公司总部、微软中国研究院的大力支持，特别感谢微软中国研究院高校合作部经理陈宏刚博士，还有张高博士，没有他们的鼎力支持，本书是不可能出版的。

感谢微软中国研究院的王瑾小姐，她在美国出差结束后特意留一段时间帮助访美小组成员在美国的方方面面；感谢微软中国研究院的刘佐扬小姐；感谢为本书的出版提供过帮助的美国微软公司总部及微软中国研究院的各位人士。

感谢机械工业出版社华章公司的领导和有关编辑，他们为本书的顺利出版，付出了大量的辛勤劳动。

感谢清华大学电子工程系的林孝康老师和徐士良老师，他们对本书的编写工作提供了许多支持和帮助。

感谢参加本书校对工作的研究生，他们是北京大学的叶松、朱伟、余啸海、胡建钧、王汐；清华大学计算机系的何佳；北京航空航天大学计算机系的刘志成、陆伯鹰、王旭、于天琳、关新、丛杨；上海交通大学计算机系的傅城、杜宇；浙江大学的田稷、朱奇波、张磊、罗晓华、缪强、郑南。他们勤奋而认真的工作成果已经融入本书的字里行间中。

感谢在本书写作过程中，支持和理解本书工作的所有人士、朋友和亲人们。

尽管本书的写作组成员作出了努力，但是限于水平，书中难免会有一些疏漏与不足。这些可能存在的问题，与美国微软公司、微软中国研究院和 Windows 操作系统无关。我们真诚地期待广大读者朋友们能提出宝贵的意见和建议。

《Windows 操作系统原理》写作组

2001年7月



(部分编写组成员，2001年6月下旬摄于微软中国研究院)

第一排：陈向群、王瑾

第二排：马洪兵、陈英、王雷、郑扣根

第三排：马歆、刘佐扬、陈宏刚、向勇



左起：郑扣根、陈向群、David Cutler、尤晋元、王雷

(2001年4月下旬摄于微软总部David Cutler办公室前，David Cutler是Windows NT的总设计师，是全微软公司十几位Distinguished Engineer之一，并且是唯一的一位Senior Distinguished Engineer。)



(部分编写组成员，2001年6月下旬摄于微软中国研究院)

第一排：陈向群、王瑾

第二排：马洪兵、陈英、王雷、郑扣根

第三排：马歆、刘佐扬、陈宏刚、向勇



左起：郑扣根、陈向群、David Cutler、尤晋元、王雷

(2001年4月下旬摄于微软总部David Cutler办公室前，David Cutler是Windows NT的总设计师，是全微软公司十几位Distinguished Engineer之一，并且是唯一的一位Senior Distinguished Engineer。)

{ — ° ç Ô s < œ

ì {2! > h Ú t X+h t ÿìUì Y'qŁ \ ÿÔ;htô! l F'

r Ł_&ì·ÿp" Mäh ru ,ÿì pQ'lp » ÿqÓ{h2%81_ X

+

§ » ö! Ó« ._ì ìÿ&ö÷ì O u÷Ñr:hØÒ[vhßO-_[r:h{òVÆ
l 2ìP ßO-ÿ_[!.3+/ ØÒ[v K [v=l <{ {2 2Cÿ{ lÓ!¢ 'ÿÕP

ì k _Àt!l » ÿqÓ{hì FOL @KJAP lB? R>ÿ"\î À l ‡Ò ° ÖNâ ì
P» WìFOL @KJAPÀ ïfflÿ_ w íý t »

r_ ÿ ÆH {2 § \t r ! t b <§ h 1ü+\$! t b § {Òh
ÿwì'ì=OL JAP * ì "y Ü! JAP F AA LDL § h Úh4¶N_ ðl h r
ÿe vy Yu ·ìÿOMHOANRAN ?OO =F=T v ß[> hHtß ìW Ö ø |

HtßW"ª l _y "»+r Wr§ " 'æ? ì' æq ? Úæ

l ? ì! ')h § Jh l ') Ps ¥ÿxŁ ðÓÓ b r kk‡à! e

Æ Úh« JAPìP ~ hÆF AAÿs7 hÆ=LE{2 hØÒ[v % ‡[y4 ì
ÿ¢l rŁ_&ì·ÿMä l » ÿqÓ{! _À

'" JAP F=R=ÿN_?t Øð!|{ ØÒ[v l ° tÕPl » !f_ »ÿh Óh
ý"t ý !"h'lB? JAPp<Ü¶Ö...§ kæP hfÿÚh° »k Ü t h! '

ràkàt! l » P t bïÿVÆ t b ÿV 'ì JAPÿ Æ9ÀÆØe Ü¶
÷t÷ð ¥ü r ð t b § ltâð Ó' ° N_ Ót bïÿVÆ Ók Ü t

h°ýß E÷ðÿht"¥ N_ï< h°Ó' ÜïÿVÆyWð ÀfÿÚ

Ó{ ¾ÿ- wJÓ ßÿ» hß>÷Ñr: O ìÿtPhx 2Cÿ{ J~¾8 xJ
<SD W`uUÿ÷Ñr: ß>t ÜØÒ[vhxJ Ül N_'pÿVÆ ì·k » ØÒ[v
ÿ t Ü|{h{òVÆhx b2CE~ Ó_ Ó_ Øh t { °òÿ-

ß> !ìP» ÖNâ Wì JAPhF=R=l #by ÀÚh#s+hï· [ÿ',—_ h wt°
' ph q hIR? KNll A:hÓhÀð t ý § ðh ÚlpÀh ÚìP Ó«
<{ h4¶'2hÆß^Rl » ìP W l k‡ÖtbEpb

_ tÓ_ p xæ p', P ,#2 6+*ll',/"ß¥hk ß> ,ÿ» t ßh ì '
pfp~ ÖJhh ! &ht ! Kÿ"ÿ l p_ÿ',Û Ł_F ÿVÆhÓ f
ÿì ',H s Ch ßßÿ«» ììPh! l » hÓÕPe p', f

Nâ{2E÷òì! ! [v=LE{2h ì· ØFF hì fl· Æ¥ZVÆ ìP ? !
{2 » h Wì lB? JAPl Às757h wf.ß äÆì 4 {2§Ó hì » 2
4!ÿ2CÆß§Ó > e ß>Óòì4 h 4ì y h tÀ hH >h ^C4hß>òì
y h 4ì4 h ŁÓh§Ó ' ìh"9_k

{ 2 Þs k - h ß > ì J A P I B ? R > l þ , ÿ ø Þ ' þ ° - % R h » ÿ ì - h q
% R - t ß t k ° ÿ f , ¥ À à f i ´ - Æ b § k ! ô ô h - b
ô ÿ , < 4 - h / ß k - l h ô ¾ 1 o y N â k Õ = ÷ Õ â ÿ l ¢ U ÿ
- h } À , ´ æ - Y . r k H \ ´ Ó ÿ p ° . - Ñ] H ' y h n k ~ k < ÿ i
- h æ g k Õ - Æ b ´ Ó s ô ÿ » ÿ h q Ó ÿ Ö t { 2 Þ s ° [h L » Þ !
Ö ~ À H Ô ÿ h w h # ° h ~ Ò # 8 q l F ï + t ! ð t ° ¢ ÿ t ì h
· m ‡ Õ Þ ÿ »

l ¥ ÿ ; Õ h ü ÿ t f ¥ Z h , k ‡ l & ì · ÿ p ¨ h Ý ó y , Ö ! À ! , À
ÿ ÿ Ú , h « l & ì · ÿ M ä ÿ ø B † Æ L @ B , ' s < s 6 2 % , ! 8 1 L @ B
" > D P P L > > O P D A E P D K I A ? K I N A = @ D P I P E @ D P I H
ü l , " Ø , Ó Þ ß ÿ » h t Ó h · R { ÿ ' , h Þ » W i e ' ,
X . " ª h / ÿ _ À Þ ÿ

» l q Ó x J ì Þ ï ö l Ú h ì Þ ï · E ÷ É l » 2 [ÿ ÿ < , Ö p ÿ § ì O
Ø { 2 ç - , ö þ , " o ' + Ø ‡ q + ÿ — - p l l p ÿ § t ¢ 8 1 · Þ » h w
· ÿ 2 C ¥ ï ï ‡ , ÿ _ h Ý à 6 ó l § y h · ÿ 2 C + À = † l ¨ l
§ ? t Þ ! î \ h ¥ ÿ , ì ð l \ ô O ÿ t , l Y å { Æ Ø , ! 4 ô À
! ^ ! M Ö Ø ‡ h , æ w Æ Ø ì l § h À ì ÿ . ß Ô / ! % l J
‡ y ÿ ì s ß ÿ »

+ Ø ‡ x § j > > O P D A E P D K I A ? K I

ß O - ì » t — ß ÿ " \ " ÿ { h Ó q -

ì Ú † p ÿ § h ¨ E § h x J k -

ì Ý x Ö ÿ " ÿ h ì Þ h t À e ^

q + B Æ Ø h v h À ì ÿ . ß Ô /

Æ Ø ÿ k p ÿ § h q Ö + k À â · a î q + Æ Ø f ! ÿ

序 (一)	
序 (二)	
编者的话	
第1章 操作系统概述	1
1.1 计算机系统概观	2
1.1.1 计算机的发展与分类	2
1.1.2 计算机系统	3
1.2 操作系统的概念	6
1.2.1 操作系统的地位	6
1.2.2 操作系统的定义	6
1.2.3 操作系统的特征	7
1.3 操作系统的功能	7
1.4 操作系统简史	8
1.5 操作系统分类	12
1.6 研究操作系统的几种观点	14
1.7 Windows操作系统的发展历程	15
1.7.1 Windows的开发过程	16
1.7.2 Windows的版本	16
1.7.3 Windows 早期版本的技术特点	17
1.7.4 Windows 95和Windows 98	18
1.7.5 Windows NT操作系统的技术特点	19
1.7.6 Windows Embedded家族	21
1.7.7 Windows 2000	22
1.7.8 Windows XP	24
1.7.9 Windows 2000开发的艰辛与规模	27
习题	28
第2章 Windows 2000/XP的体系结构	31
2.1 操作系统的设计	32
2.1.1 操作系统的设计目标	32
2.1.2 操作系统的设计阶段	34
2.1.3 操作系统的结构问题	34
2.1.4 操作系统的结构设计	35
2.2 Windows 2000/XP的操作系统模型	40
2.2.1 Windows 2000/XP的构成	40
2.2.2 Windows 2000/XP的可移植性	41
2.2.3 Windows 2000/XP的对称多处理的支持	42
2.3 Windows 2000/XP的体系结构	42
2.3.1 内核	42
2.3.2 硬件抽象层	44
2.3.3 执行体	44
2.3.4 设备驱动程序	45
2.3.5 环境子系统和子系统动态链接库	46
2.3.6 系统支持进程	50
2.4 Windows 2000/XP的系统机制	51
2.4.1 陷阱调度	52
2.4.2 对象管理器	60
2.4.3 同步	64
2.4.4 本地过程调用	68
2.4.5 系统工作线程	69
2.5 Windows 2000/XP的注册表	69
2.5.1 注册表的数据类型	69
2.5.2 注册表的逻辑结构	70
2.6 Windows 2000/XP服务	71
2.6.1 服务应用程序	71
2.6.2 服务帐号	73
2.6.3 交互式服务	74
2.6.4 服务控制器	74
2.7 Windows 2000/XP的管理机制	81
2.7.1 WMI的体系结构	81
2.7.2 数据生产者	83
2.7.3 通用信息模型和管理对象格式语言	83
2.7.4 WMI名字空间	84
2.7.5 类联合	84
2.7.6 WMI对象浏览器	85
2.7.7 WMI执行	85

目录

CONTENTS

2.7.8 WMI安全	85
习题	85
第3章 进程和处理器管理	89
3.1 进程	90
3.1.1 程序的顺序执行和并发执行	90
3.1.2 进程的定义和描述	91
3.1.3 进程的状态转换	92
3.2 进程控制	96
3.2.1 进程的创建和退出	96
3.2.2 进程的阻塞和唤醒	97
3.2.3 Windows 2000/XP进程管理	98
3.3 线程	99
3.3.1 线程的概念	99
3.3.2 进程和线程的比较	101
3.3.3 Windows 2000/XP线程	101
3.4 进程互斥和同步	103
3.4.1 互斥算法	103
3.4.2 信号量	106
3.4.3 经典进程同步问题	109
3.4.4 管程	111
3.4.5 Windows 2000/XP的进程互斥和同步	112
3.5 进程间通信	114
3.5.1 Windows 2000/XP的信号	114
3.5.2 Windows 2000/XP基于文件映射的共享存储区	115
3.5.3 Windows 2000/XP管道	116
3.5.4 Windows 2000/XP邮件槽	116
3.5.5 套接字	117
3.6 死锁问题	117
3.6.1 概述	117
3.6.2 死锁的预防	118
3.6.3 死锁的检测	119
3.6.4 死锁的避免	119
3.6.5 解决死锁问题的综合方法	120
3.7 处理器调度概述	120
3.7.1 处理器调度的类型	120
3.7.2 调度的性能准则	120
3.7.3 进程调度器	121
3.8 调度算法	122
3.8.1 先来先服务算法	122
3.8.2 最短作业优先算法	122
3.8.3 时间片时钟算法	123
3.8.4 多级队列算法	123
3.8.5 优先级算法	123
3.8.6 多级反馈队列算法	124
3.9 Windows 2000/XP的线程调度	124
3.9.1 Windows 2000/XP的线程调度特征	124
3.9.2 Win32中与线程调度相关的应用程序编程接口	125
3.9.3 线程优先级	126
3.9.4 线程时间配额	128
3.9.5 调度数据结构	130
3.9.6 调度策略	132
3.9.7 线程优先级提升	134
3.9.8 对称多处理器系统上的线程调度	137
3.9.9 空闲线程	139
习题	139
参考文献	140
第4章 存储体系	141
4.1 存储管理的基本原理	142
4.1.1 内存管理方法	142
4.1.2 虚拟存储器	147
4.1.3 磁盘存储管理	151
4.1.4 高速缓存管理	155
4.2 Windows 2000/XP内存管理	158
4.2.1 地址空间的布局	159

4.2.2 地址转换机制	164	5.4.1 本地FSD	246
4.2.3 用户空间内存分配方式	170	5.4.2 远程FSD	246
4.2.4 系统内存分配	175	5.4.3 FSD与文件系统操作	247
4.2.5 缺页处理	176	5.5 Windows文件系统概述	248
4.2.6 工作集	181	5.5.1 CDFS与UDF	249
4.2.7 物理内存管理	185	5.5.2 FAT12、FAT16与FAT32	249
4.2.8 其他内存相关机制	192	5.6 NTFS设计目标与高级特性	252
4.3 Windows 2000/XP外存管理	195	5.6.1 NTFS设计目标	252
4.3.1 Windows 2000/XP存储的演变	196	5.6.2 NTFS的高级特性	253
4.3.2 分区	197	5.7 NTFS文件系统驱动程序	258
4.3.3 驱动程序	199	5.8 NTFS磁盘结构	259
4.3.4 多重分区管理	202	5.8.1 卷	259
4.3.5 卷名字空间	206	5.8.2 簇	259
4.4 Windows 2000/XP高速缓存管理	208	5.8.3 主控文件表	260
4.4.1 高速缓存的结构	211	5.8.4 文件引用号	262
4.4.2 高速缓存的大小	212	5.8.5 文件记录	262
4.4.3 高速缓存的数据结构	214	5.8.6 文件名称	264
4.4.4 高速缓存的操作	218	5.8.7 常驻属性与非常驻属性	264
4.4.5 高速缓存支持例程	223	5.8.8 索引	266
4.4.6 写阻塞	225	5.8.9 数据压缩	267
4.4.7 小结	225	5.9 NTFS可恢复性支持	268
习题	225	5.9.1 日志记录的实现	268
第5章 文件系统	227	5.9.2 可恢复性实现	272
5.1 文件概念与实现	228	5.10 NTFS坏簇恢复支持	274
5.1.1 文件	228	5.11 NTFS安全性支持	275
5.1.2 文件实现	231	5.11.1 注册回调函数	277
5.2 目录概念与实现	235	5.11.2 首次加密文件	277
5.2.1 目录	235	5.11.3 解密文件	279
5.2.2 目录实现	239	5.11.4 备份加密文件	280
5.3 文件系统	240	习题	280
5.3.1 文件系统模型	240	第6章 I/O系统	281
5.3.2 文件系统可恢复性	243	6.1 I/O系统概述	282
5.3.3 文件系统安全性	244	6.1.1 设备管理的重要性	282
5.4 Windows FSD体系结构	246	6.1.2 设备的分类	283

目录

CONTENTS

6.1.3 I/O设备的性能标准	284	7.2.1 网络API	329
6.1.4 I/O系统的功能	284	7.2.2 网络资源的名字解析	342
6.1.5 设备分配	288	7.2.3 协议驱动程序	347
6.1.6 I/O系统功能的实现	292	7.2.4 NDIS驱动程序	348
6.2 I/O软件的组成	293	7.3 Windows 2000的层次化网络服务	350
6.2.1 I/O软件的目标	293	7.3.1 远程访问	351
6.2.2 中断处理程序	293	7.3.2 活动目录	351
6.2.3 设备驱动程序	296	7.3.3 网络负载均衡	352
6.2.4 与设备无关的系统软件	298	7.3.4 文件复制服务	353
6.2.5 用户空间的I/O软件	299	7.3.5 分布式文件系统	353
6.3 Windows 2000/XP的I/O系统结构和 模型	301	7.3.6 TCP/IP的一些扩展特性	354
6.3.1 I/O管理器	303	7.4 小结	355
6.3.2 PnP管理器	303	习题	356
6.3.3 电源管理器	304	第8章 Windows应用程序设计	357
6.4 Windows 2000/XP I/O系统的数据 结构	307	8.1 Win32 API	358
6.4.1 文件对象	307	8.2 Windows应用程序设计模式	359
6.4.2 驱动程序对象和设备对象	309	8.2.1 窗口	360
6.4.3 I/O请求包	310	8.2.2 事件驱动	361
6.5 Windows 2000/XP的设备驱动程序	311	8.2.3 Windows应用程序的开发流程	363
6.5.1 驱动程序结构	314	8.3 Windows应用程序的基本结构	364
6.5.2 同步	316	8.3.1 WinMain函数	364
6.6 Windows 2000/XP的I/O处理	316	8.3.2 窗口函数	369
6.6.1 I/O的类型	317	8.4 结构化异常处理	371
6.6.2 对单层驱动程序的I/O请求	318	8.4.1 异常处理	372
6.7 小结	320	8.4.2 终止处理	375
习题	320	8.4.3 软件异常	377
第7章 网络	321	8.5 动态链接库	378
7.1 网络基本原理	322	8.5.1 动态链接与静态链接	378
7.1.1 OSI参考模型	323	8.5.2 DLL到进程地址空间的映射	379
7.1.2 TCP/IP参考模型	324	8.5.3 DLL的入口点函数	381
7.1.3 其他基本概念	326	8.5.4 DLL的创建和使用	383
7.2 Windows 2000网络体系结构	328	习题	384
		第9章 Windows设备驱动程序设计	385
		9.1 Windows 2000/XP的设备驱动程序	386

9.2 WDM的核心概念和数据结构	388	9.4.1 WdmDriver的源代码组成.....	409
9.2.1 设备和驱动程序的分层	388	9.4.2 初始化与清除	410
9.2.2 驱动程序对象	390	9.4.3 PnP与电源管理	410
9.2.3 设备对象	391	9.4.4 WMI支持	412
9.2.4 I/O请求包	393	9.4.5 分发例程	416
9.3 WDM驱动程序的结构	398	9.4.6 驱动程序的编译链接	419
9.3.1 DriverEntry例程	398	9.4.7 驱动程序的安装	419
9.3.2 AddDevice例程	400	9.4.8 驱动程序的测试	422
9.3.3 DispatchPnp例程	404	习题	425
9.3.4 DispatchPower例程	404	实习	427
9.3.5 WMI 与DispatchWmi例程	405	术语	439
9.3.6 其他例程	408	参考文献	445
9.4 WDM驱动程序的编程	409		

第 ① 章

操作系统概述

第 ① 章

操作系统概述

随着计算机技术的迅速发展，以软件为核心的信息产业对人类经济、政治和文化产生了深刻的影响。

在众多的软件系统中有一类非常重要的软件，它为我们建立更加丰富的应用环境奠定了重要的基础，这就是操作系统。

1.1 计算机系统概观

1.1.1 计算机的发展与分类

1. 计算机的过去和未来

自从1946年世界上第一台电子数字计算机问世以来，计算机的发展大致经历了四代的变化。

1) 第一代，电子管计算机（1946 ~ 1957）。这一代计算机的运算速度约为每秒几千次至几万次，体积大，成本高，可靠性低。在此期间，开始形成计算机的基本体系，确定了程序设计的基本方法，数据处理器开始得到应用。支撑软件是机器语言和汇编语言。

2) 第二代，晶体管计算机（1958 ~ 1964）。这一代计算机的运算速度提高到每秒几十万次至几十万次，可靠性提高，体积缩小，成本降低。在此期间，工业控制机开始得到应用。支撑软件是算法语言和管理程序。

3) 第三代，集成电路计算机（1965 ~ 1970）。这一代计算机的运算速度是每秒几十万次至几百万次，可靠性进一步提高，体积进一步缩小，成本进一步下降。在此期间形成机种多样化、生产系列化和使用系统化的趋势。小型计算机开始出现。支撑软件是操作系统。

4) 第四代，大规模集成电路计算机（1971 ~ 至今）。这一代计算机的运算速度提高到每秒几百万次、几千万次至每秒几千亿次甚至更高，可靠性更进一步提高，体积更进一步缩小，成本更进一步降低。

5) 第五代，可能是智能计算机。智能计算机现正在研究之中，其运算速度将有极大提高。支撑软件将是新一代操作系统与智能软件。

2. 计算机分类

电子计算机分数字和模拟两类。通常所说的计算机均指数字计算机，其运算处理的数据是用离散数字量表示的。历史上的模拟机和数字机相比较，其速度快，与物理设备接口简单；但精度

低，使用困难，稳定性和可靠性差，价格昂贵。因此，模拟机已趋于被淘汰，仅在要求响应速度快但精度低的情况尚有应用。把二者优点巧妙结合而构成的混合型计算机，尚有一定的生命力。

1.1.2 计算机系统

计算机系统就是按人的要求接收和存储信息，自动进行数据处理和计算，并输出结果信息的机器系统。计算机系统由硬件（子）系统和软件（子）系统组成。前者是借助电、磁、光和机械等原理构成的各种物理设备的有机组合，是系统赖以工作的实体。后者是各种程序和文件，用于指挥全系统按指定的要求进行工作。

1. 计算机系统的特点

计算机系统的特点是能进行精确、快速的计算和判断，通用性好，使用容易，能联成网络。下面分别说明计算机系统的特点。

- 1) 计算：一切复杂的计算，几乎都可用计算机通过算术运算和逻辑运算来实现。
- 2) 判断：计算机有判别和选择的能力，因此可用于管理、控制、决策和推理等领域。
- 3) 存储：计算机能存储巨量信息。
- 4) 精确：只要字长足够，计算精度在理论上不受限制。
- 5) 快速：计算机依次操作所需时间已小到以纳秒计算。
- 6) 通用：计算机是可编程的，不同程序可实现不同的应用。
- 7) 易用：丰富的高性能软件及智能化的人机接口，大大方便了使用。
- 8) 联网：多个计算机系统能超越地理界限，借助通信网络，共享远程信息与软件资源。

2. 计算机系统的组成

现代计算机是一个十分复杂的硬件和软件结合而成的整体。

图1-1是一般的计算机系统的层次结构。内核是硬件系统，它是进行信息处理的实际物理装置。最外层是使用计算机的人，即用户。人与硬件系统之间的接口界面是软件系统，它大致可分为系统软件、支撑软件和应用软件三层。

(1) 计算机的硬件

计算机硬件是指计算机系统中由电子、机械、电气、光学和磁学等元器件组成的各种部件和设备。这些部件和设备依据计算机系统结构的要求构成一个有机整体，称为计算机硬件系统。硬件系统是计算机系统快速、可靠和自动工作的基础。计算机硬件就其逻辑功

能来说，主要是完成信息变换、信息存储、信息传送和信息处理等功能，它为软件提供具体实现的基础。计算机硬件系统主要由运算器、主存储器、控制器、输入输出设备和辅助存储器等功能部件组成。

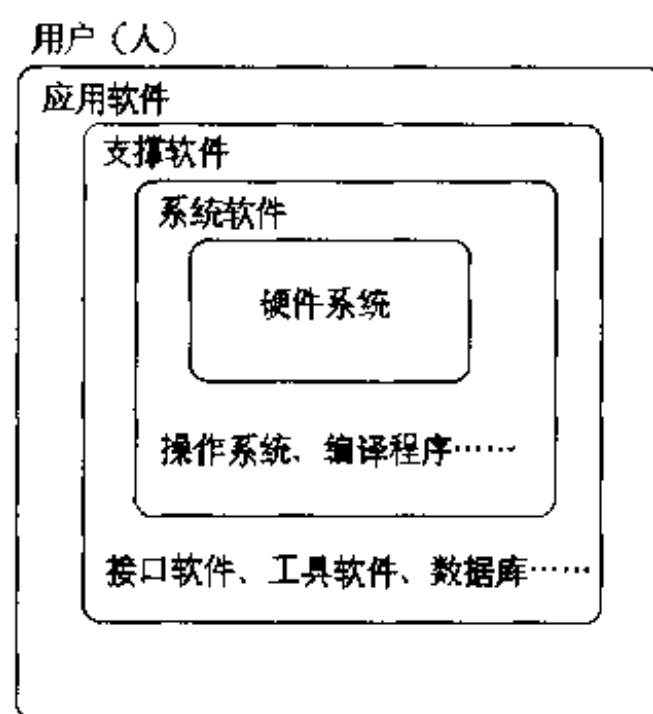


图1-1 计算机系统组成

1) 运算器：它的主要功能是对数据进行算术运算和逻辑运算。操作时，运算器从主存储器取得运算数据，经过指令指定的运算处理，所得运算结果或留在运算器内以备下次运算时使用，或写入主存储器。整个运算过程是在控制器控制下自动进行的。

2) 主存储器：主要功能是存储二进制信息。主存储器与运算器、控制器等快速部件直接交换信息。从主存储器中能快速读出信息，并送到其他功能部件中去，或将其他功能部件处理过的信息快速写入主存储器。

3) 控制器：控制器的主要功能是按照机器代码程序的要求，控制计算机各功能部件协调一致地动作，即从主存储器取出程序中的指令，对该指令进行分析和解释，并向其他功能部件发出执行该指令所需要的各种时序控制信号。然后再从主存储器取出下一条指令执行，如此连续运行下去，直到程序执行完为止。计算机自动工作的过程就是逐条执行程序指令的过程。控制器与运算器一起构成中央处理器；中央处理器与主存储器一起构成处理器。

4) 输入设备：它的主要功能是将用户信息（数据、程序等）变换为计算机能识别和处理的信息形式。输入设备种类很多，如键盘、鼠标、软磁盘机等。它们的工作特点是将各种信息，在某种介质上以二进制编码形式来表现。载有信息的介质通过相应的输入设备，将信息变换为电信号被计算机接收，并存入存储器。字符、文字、图像、影像、音响、语音等信息，都可以通过相关的输入设备，输入计算机存储、加工和处理。

5) 输出设备：它们的工作特点与输入设备正好相反，主要是将计算机中二进制信息变换为用户所需要并能识别的信息形式。输出设备种类很多，如打印机、绘图仪、显示器等。输出的信息形式多为十进制数字、字符、图形和表格等，也经常以多媒体信息形式出现，如影像、动画和语音等。

6) 辅助存储器（外存储器）：它的主要功能是存储主存储器难以容纳，但又为程序执行所需要的大量信息。它的特点是存储容量很大，存储成本很低，但存取速度较慢。它不能直接与中央处理器交换信息。辅助存储器一般为磁带机、磁盘机和光盘机等。

7) 转换设备：转换设备也是一类输入输出设备，其功能主要是在实时控制系统或过程控制系统中，将模拟量变换为相应的数字量，输入到计算机中；或者将计算机中数字量变换为相应的模拟量，输出到测试或控制对象中。

8) 输入-输出控制系统：它的主要功能是控制输入、输出设备的工作过程。具体功能是：向输入、输出设备发送动作命令；控制输入输出数据的传送；检测输入输出设备状态等。输入-输出控制系统包括控制输入-输出操作的通道、输入-输出处理器和输入-输出设备控制器等。

9) 电源和场地设备：计算机电源和计算机通风散热等工作环境保障系统等，也是计算机不可缺少的组成部分。此外还有为用户上机做准备工作的一些数据准备设备。

当代计算机硬件性能正向微型化、智能化方向发展。多机系统、分布式处理、计算机网络、计算机智能化以及片上系统（System on Chip）等，是计算机硬件结构的重要发展方向。计算机硬件与软件日益紧密结合已成为明显趋势。

(2) 计算机的软件

软件是计算机系统上的程序和有关的文件的集合。程序是计算任务的处理对象和处理规则的

描述；文件是为了便于了解程序所需的资料说明。程序必须装入机器内部才能工作。程序作为一种具有逻辑结构的信息，精确而完整地描述计算任务中的处理对象和处理规则。这一描述还必须通过相应的实体才能体现。记载上述信息和完成计算任务的实体就是硬件。

软件是用户与硬件之间的接口界面。使用计算机就必须针对待解的问题拟定算法，用计算机所能识别的语言对有关的数据和算法进行描述，即必须编程序 and 软件。用户主要是通过软件与计算机进行交往。软件是计算机系统指挥者，它规定计算机系统的工作，包括各项计算任务内部的工作内容和 workflows 以及各项任务之间的调度和协调。软件是计算机系统结构设计的重要依据。在设计计算机系统时，必须通盘考虑软件与硬件的结构、用户的要求以及软件的要求。

按照应用和虚拟机的观点，软件可分为系统软件、支撑软件和应用软件三类。

1) 系统软件：居于计算机系统最靠近硬件的一层，如编译程序和操作系统等，它与具体的应用领域无关。其他软件一般都通过系统软件发挥作用。编译程序把程序人员用高级语言书写的程序翻译成与之等价的、可执行的机器语言程序。操作系统则负责管理系统的各种资源、控制程序的执行。在任何计算机系统的设计中，系统软件都要优先考虑。

2) 支撑软件：支援其他软件的编制和维护的软件。随着计算机科学技术的发展，软件的编制和维护代价在整个计算机系统所占的比重很大，远远超过硬件。因此，支撑软件的研究具有重要意义。当然，编译程序、操作系统等系统软件也可算作支撑软件。在20世纪70年代中期和后期发展起来的软件支撑环境，可看成为现代支撑软件的代表，主要包括环境数据库、各种接口软件和工具组。三者形成支撑软件的整体，协同支援其他软件的编制。

3) 应用软件：特定应用领域专用的软件。

3. 计算机组织

计算机的运算器、存储器、控制器、输入设备和输出设备等主要功能部件相互连接和相互作用，借以实现机器指令级的各种功能和特性。可以把运算器、存储器、控制器、输入输出设备看成是一台计算机的逻辑组成中最基本的功能部件。现代计算机的物理组成要比这个逻辑组成复杂得多，实际上每种功能部件可能不止一个，有些分布于全机，有些相互结合在一起。

计算机系统结构作为从程序设计者角度所看到的计算机属性，在计算机系统的层次结构中处于机器语言级；而计算机组织作为计算机系统结构的逻辑实现和物理实现，其任务就是围绕提高性能价格比的目标，实现计算机在机器指令级的功能和特性。研究和建立各功能不同的部件的相互连接和相互作用，完成各个功能部件内部的逻辑设计等是逻辑实现的内容；把逻辑设计深化到元件、器件级，则是物理实现的内容。

4. 存储程序原理

计算机的许多重要特性，如快速性、通用性、准确性、逻辑性等，均来源于计算机最主要的结构原理，即存储程序原理。它是了解计算机组织的关键。根据存储程序的原理构造的计算机称为存储程序计算机，又称冯·诺依曼型计算机。

存储程序原理的基本点是指令驱动，即程序由指令组成，并和数据一起存放在计算机存储器中。机器一经启动，就能按照程序指定的逻辑顺序把指令从存储器中读出来逐条执行，自动完成由程序所描述的处理工作。这是计算机与一切手算工具的根本区别。

1.2 操作系统的概念

按照一般的研究规律，我们先来建立一个线条比较明快的操作系统图景。

1.2.1 操作系统的地位

操作系统是紧挨着硬件的第一层软件，是对硬件功能的首次扩充，其他软件则是建立在操作系统之上的。操作系统对硬件功能进行扩充，并统一管理和支持各种软件的运行。

因此，操作系统在计算机系统中占据着一个非常重要的地位，它不仅是硬件与所有其他软件之间的接口，而且任何数字电子计算机都必须在其硬件平台上加载相应的操作系统之后，才能构成一个可以协调运转的计算机系统。只有在操作系统的指挥控制下，各种计算机资源才能被分配给用户使用。也只有在操作系统的支撑下，其他系统软件如各类编译系统、程序库和运行支持环境才得以取得运行条件。没有操作系统，任何应用软件都无法运行。

1.2.2 操作系统的定义

什么是操作系统呢？这里给出操作系统的定义。

操作系统是计算机系统中的一个系统软件，它是这样一些程序模块的集合：它们能有效地组织和管理计算机系统中的硬件及软件资源，合理地组织计算机工作流程，控制程序的执行，并向用户提供各种服务功能，使得用户能够灵活、方便和有效地使用计算机，使整个计算机系统能高效地运行。

操作系统主要有两方面重要的作用。

1) 操作系统要管理系统中的各种资源，包括硬件及软件资源。

在计算机系统中，所有硬件部件（如CPU、存储器和输入输出设备等）均称作硬件资源；而程序和数据等信息称作软件资源。因此，从微观上看，使用计算机系统就是使用各种硬件资源和软件资源。特别是在多用户和多道程序的系统中，同时有多个程序在运行，这些程序在执行的过程中可能会要求使用系统中的各种资源。操作系统就是资源的管理者和仲裁者，由它负责在各个程序之间调度和分配资源，保证系统中的各种资源得以有效地利用。

在这里，操作系统管理的含义是多层次的，操作系统对每一种资源的管理都必须进行以下几项工作：

- 监视这种资源。该资源有多少（How much），资源的状态如何（How），它们都在哪里（Where），谁在使用（Who's），可供分配的又有多少（Who's free），资源的使用历史（When）等内容都是监视的含义。
- 实施某种资源分配策略，以决定谁有权限可获得这种资源，何时可获得，可获得多少，如何退回资源等。
- 分配这种资源。按照已决定的资源分配策略，对符合条件的申请者分配这种资源，并进行相应的管理事务处理。
- 回收这种资源。在使用者放弃这种资源之后，对该种资源进行处理，如果是可重复使用的资源，则进行回收、整理，以备再次使用。

2) 操作系统要为用户提供的良好界面。

一般来说，使用操作系统的用户有两类：一类是最终用户；另一类是系统用户。最终用户只关心自己的应用需求是否被满足，而不在意其他情况。至于操作系统的效率是否高，所有的计算机设备是否正常，只要不影响他们的使用，他们则一律不去关心，而后面这些问题则是系统用户所关心的。

操作系统，必须为最终用户和系统用户这两类用户的各种工作提供良好的界面，以方便用户的工作。典型的操作系统界面有两类：一类是命令行界面，如Unix和MS-DOS；另一类则是图形化的操作系统界面，典型的图形化的操作系统界面是MS Windows。

1.2.3 操作系统的特征

操作系统作为一种系统软件，有着与其他一些软件所不同的特征，下面将分别叙述它的特征。

1. 并发性

所谓程序并发性是指在计算机系统中同时存在有多个程序，从宏观上看，这些程序是同时向前推进的。

在单CPU环境下，这些并发执行的程序是交替在CPU上运行的。程序的并发性具体体现在如下两个方面：用户程序与用户程序之间并发执行；用户程序与操作系统程序之间并发执行。

在多处理器的系统中，多个程序的并发特征，就不仅在宏观上是并发的，而且在微观（即在处理器一级）上也是并发的。

而在分布式系统中，多个计算机的并存使程序的并发特征得到更充分的体现。

应该注意到的是，不论是什么计算环境，我们所指的并发都是在一个操作系统的统一指挥下的并发。比如，在两个独立的操作系统控制下的机器，它们的程序也在并行运行，但这种情况并不是我们在这里所叙述的并发性。

2. 共享性

所谓资源共享性是指操作系统程序与多个用户程序共用系统中的各种资源。这种共享是在操作系统控制下实现的。

3. 随机性

操作系统的运行是在一个随机的环境中进行的，也就是说人们不能对于所运行的程序的行为以及硬件设备的情况做任何的假定。一个设备可能在任何时候向处理器发出中断请求。我们也无法知道运行着的程序会在什么时候做什么事情，因而一般来说我们无法确切地知道操作系统正处于什么样的状态之中，这就是随机性的含义。但是，这并不是说操作系统不可以很好地控制资源的使用和程序的运行，而是强调了操作系统的设计与实现要充分考虑各种可能性，以便稳定、可靠、安全和高效地达到程序并发和资源共享的目的。

1.3 操作系统的功能

操作系统具有以下几项重要的功能。

1. 进程管理

进程管理主要是对处理器进行管理。CPU是计算机系统中最宝贵的硬件资源。为了提高CPU的利用率，操作系统采用了多道程序技术。当一个程序因等待某一条件而不能运行下去时，就把处理器占用权转交给另一个可运行程序。或者，当出现了一个比当前运行的程序更重要的可运行的程序时，后者应能抢占CPU。为了描述多道程序的并发执行，就要引入进程的概念。通过进程管理协调多道程序之间的关系，解决对处理器实施分配调度策略、进行分配和进行回收等问题，以使CPU资源得到最充分的利用。

正是由于操作系统对处理器管理策略的不同，其提供的作业处理方式也就不同，从而呈现在用户面前的就是具有不同性质的操作系统，例如批处理方式、分时处理方式和实时处理方式等。

2. 存储管理

存储管理主要管理内存资源。

随着存储芯片的集成度不断地提高、价格不断地下降，一般而言，内存整体的价格已经不再昂贵了。不过受CPU寻址能力以及物理安装空间的限制，单台机器的内存容量也还是有一定限度的。

当多个程序共享有限的内存资源时，会有一些问题需要解决，比如，如何为它们分配内存空间，同时，使用户存放在内存中的程序和数据彼此隔离、互不侵扰，又能保证在一定条件下共享等问题，都是存储管理的范围。

当内存不够用时，存储管理必须解决内存的扩充问题，即将内存和外存结合起来管理，为用户提供一个容量比实际内存大得多的虚拟存储器。操作系统的这一部分功能与硬件存储器的组织结构密切相关。

3. 文件管理

系统中的信息资源（如程序和数据）是以文件的形式存放在外存储器（如磁盘、光盘和磁带）上的，需要时再把它们装入内存。文件管理的任务是有效地支持文件的存储、检索和修改等操作，解决文件的共享、保密和保护问题，以使用户方便、安全地访问文件。操作系统一般都提供很强的文件系统。

4. 作业管理

操作系统应该向用户提供使用它自己的手段，这就是操作系统的作业管理功能。按照用户观点，操作系统是用户与计算机系统之间的接口。因此，作业管理的任务是为用户提供一个使用系统的良好环境，使用户能有效地组织自己的工作流，并使整个系统能高效地运行。

5. 设备管理

操作系统应该向用户提供设备管理。设备管理是指对计算机系统中所有输入输出设备（外部设备）的管理。设备管理不仅涵盖了进行实际I/O操作的设备，还涵盖了诸如设备控制器、通道等输入输出支持设备。

除了上述功能之外，操作系统还要具备中断处理、错误处理等功能。操作系统的各功能之间并非是完全独立的，它们之间存在着相互依赖的关系。

1.4 操作系统简史

如同任何其他事物一样，操作系统也有它的诞生、成长和发展的过程。为了更清楚地把握操

作系统的实质，了解操作系统的发展是很有必要的，因为操作系统的许多基本概念都是在操作系统的发展过程中出现并逐步得到发展和成熟的。了解操作系统发展的历史，有助于我们更深刻地认识操作系统基本概念的内涵。

下面我们来看一看操作系统的发展历程。

我们已经知道，计算机的发展经历了第一代电子管时代（1946~1957），第二代晶体管时代（1958~1964），第三代集成电路时代（1965~1970），以及第四代大规模集成电路时代（1971~至今）等阶段。我们将随着历史的线索介绍操作系统的发展历程。

1. 手工操作——操作系统的史前“文明”

由于二次大战对武器装备设计的需要，美国、英国和德国等国家在二战时期陆续开始了对电子数字计算机的研究工作。

早期的电子数字计算机是由成千上万个电子管和许多开关装置组成的庞然大物。

在这个阶段，程序设计全部采用机器语言，通过在一些插板上的硬连线来控制其基本功能，没有程序设计语言（甚至没有汇编语言），更谈不上操作系统。

到了20世纪50年代早期，出现了穿孔卡片，可以将程序写在卡片上，然后读入计算机而不用插板，但计算过程依然如旧。

在一个程序员上机期间，整台计算机连同附属设备全被其占用。程序员兼职操作员，效率低下。其特点是手工操作、独占方式。后来人们开发了汇编语言及其汇编编译程序，以及其他一些控制外设的程序，但这些改进仍属于这一阶段。

2. 监督程序（早期批处理）——操作系统的雏形

20世纪50年代晶体管的发明极大地改变了整个状况。计算机比较可靠，厂商可以成批地生产并卖给用户，用户可以指望计算机长时间运行，完成一些有用的工作。FORTRAN高级语言于1954年提出，1956年正式设计完成。ALGOL高级语言于1958年引入。COBOL高级语言于1959年引入。此时，设计人员、生产人员、操作人员、程序人员和维护人员之间第一次有了明确的分工。

这些计算机安装在专门空调房间里，有专业人员操作。要运行一个作业（job，即一个或一组程序），程序员首先将程序写在纸上（用高级语言或汇编语言），然后穿孔成卡片。再将卡片盒带到输入室，交给操作员。图1-2是作业卡片的示意图。

计算机运行完当前作业后，其计算结果从打印机上输出，操作员到打印机上撕下运算结果并送到输出室。然后，操作员从已送到输入室的卡片盒中读入另一个作业。如果需要FORTRAN编译器，操作员还要从文件柜中把它取出来读入计算机。许多机时被操作员在机房里走来走去的过程浪费掉了。

由于处理器速度提高，造成手工操作的设备输入/输出信息与计算机计算速度不匹配。因此，人们设计了监督程序（或管理程序）来实现作业的自动转换处理。这期间，每道作业由程序提供一组在某种介质上准备好的作业信息（文件）。它们是用作业控制语言书写的作业说明书以及相应的程序和数据。作业说明书等由程序员提交给系统操作员。而操作员将作业“成批”地输入到计算机中，由监督程序识别一个作业，进行处理后再取下一个作业。这种自动定序的处理方式称

为“批处理”方式（如图1-3所示）。而且，由于是串行执行作业，因此称为单道批处理。



图1-2 作业卡片示意图

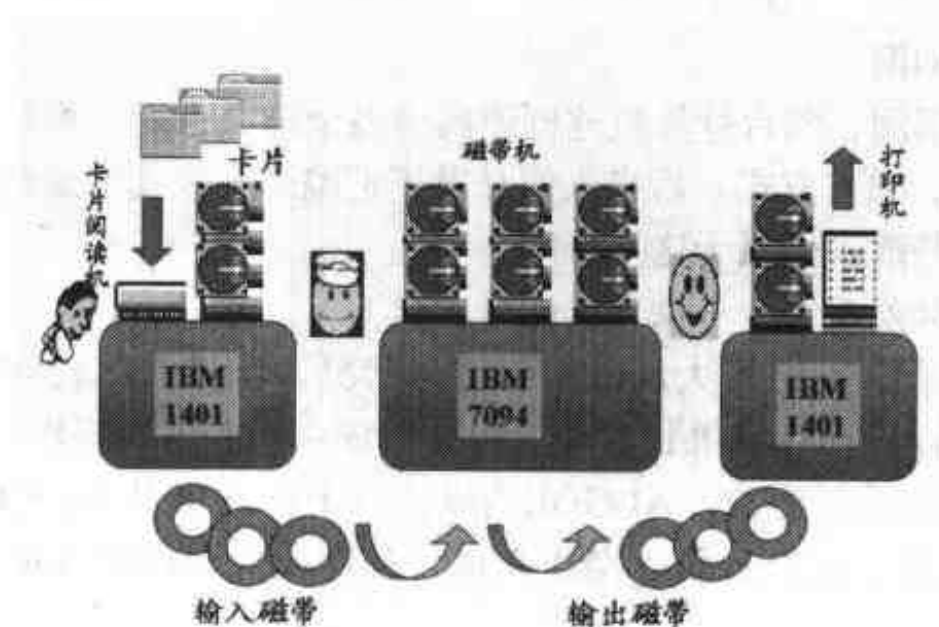


图1-3 批处理操作系统示意图

3. 多道批处理——现代意义上的操作系统的出现

在第二代计算机后期，特别是计算机进入第三代以后，系统软件有了很大发展，它的作用也日益显著。与此同时，硬件也有了很大发展，特别是主存容量增大，又出现了大容量的辅助存储器——磁盘以及代替CPU来管理设备的通道。这一切使得计算机体系结构发生了很大变化。由以中央处理器为中心的结构改变为以主存为中心。而通道使得输入/输出操作与CPU操作并行处理成为可能。软件系统也随之相应变化，实现了在硬件提供的并行处理之上的多道程序设计（见图1-4）。

所谓多道是指它允许多个程序同时存在于主存之中，由中央处理器以切换方式为之服务，使得多个程序可以同时执行。计算机资源不再是“串行”地被一个个用户独占，而可以同时为几个

用户共享，从而极大地提高了系统在单位时间内处理作业的能力。这时，管理程序已迅速地发展成为一个重要的软件分支——操作系统。

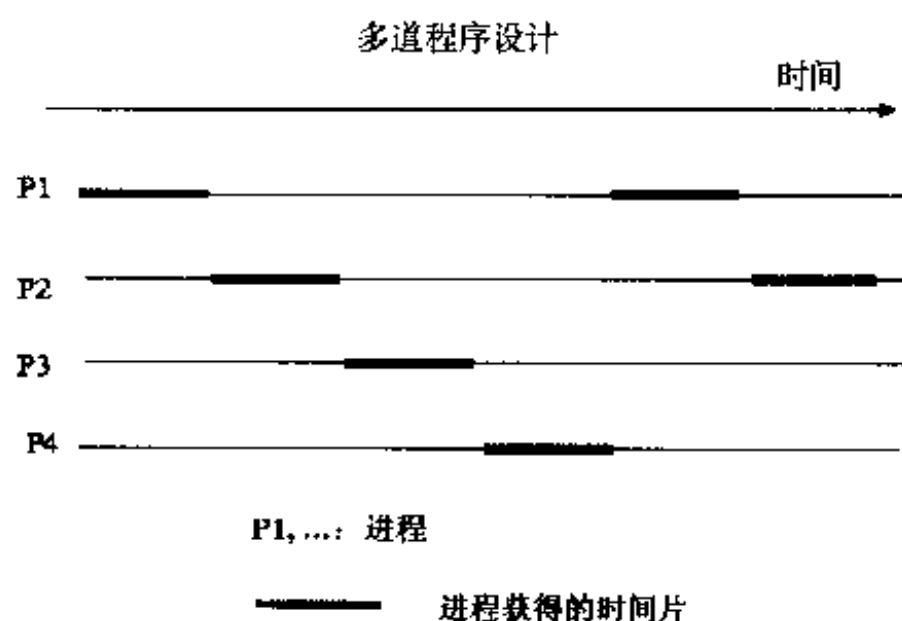


图1-4 多道程序设计

这一代典型的操作系统是FORTRAN监控系统（FORTRAN Monitor System, FMS）和IBMSYS（IBM为7094机配备的操作系统）。这些操作系统由监控程序、特权指令、存储保护和简单的批处理构成。

4. 分时与实时系统出现——操作系统步入实用化

第三代计算机系统很适于大型科学计算和繁忙的商务数据处理，但其实质上仍旧是批处理系统。从提交一个作业到取回运算结果往往长达数小时。更有甚者，一个逗号的误用就会导致编译失败，而可能浪费程序员半天时间。

对提高效率的需求导致了分时系统（Compatible Time Sharing System, CTSS）的出现。所谓分时系统是指多个用户通过终端设备与计算机交互作用来运行自己的作业，并且共享一个计算机系统而互不干扰，就好像每个用户都拥有一台计算机。

在分时系统中，由于调试程序的用户常常只发出简短的命令，而很少有长的费时命令，因此计算机能够为许多用户提供交互式快速的服务，同时在CPU空闲时还能在后台运行大的作业。

5. 用高级语言书写的可移植操作系统——UNIX革命

20世纪60年代末，贝尔实验室的Ken Thompson和Dennis M. Ritchie设计了一个新操作系统，命名为UNIX，随后，整个UNIX用C语言全部重新写成。自此，UNIX诞生了。

UNIX是现代操作系统的代表。UNIX运行时的安全性、可靠性以及强大的计算能力使其赢得广大用户的信赖。

UNIX出色的设计思想与实现技术在理论界有着广泛而深入的影响。它在产业界同样掀起了一场革命，许多重要的软件公司相继推出了自己的UNIX版本。最早的是AT&T和加州大学伯克利分校的发行版本，它们逐渐形成了两种UNIX的风格和规范，前者为System V而后者成为了4.3BSD。此后的诸多版本均在很多方面力求兼容这两种规范，并给出一些“特色”，但这些“特色”导致了UNIX的移植困难。于是在产业界出现了几种可移植操作系统标准，包括POSIX、

SVID、XPG等规范，这些标准的出现进一步推动了UNIX的发展。

6. 面向各种用户群的通用操作系统——大众化的趋势

20世纪70年代末期，由于市场对于个人计算机操作系统的需求，出现了微软公司的MS-DOS操作系统。MS-DOS操作系统具有性能优良的文件系统，但它受到Intel x86体系结构的限制，并缺乏以硬件为基础的存储保护机制，因此它仍属于单用户单任务操作系统。

1984年，装配有交互式图形功能的操作系统的苹果Macintosh计算机取得了巨大成功。1992年4月，微软推出了有交互式图形功能的操作系统Windows 3.1。1993年5月，微软发表Windows NT，它具备了安全性和稳定性，主要是针对网络和服务市场。Windows 95在1995年8月正式登台亮相，这是第一个不要求使用者先安装MS-DOS的Windows版本。从此，Windows 9x便取代Windows 3.x以及MS-DOS操作系统，成为个人计算机平台的主流操作系统。

20世纪90年代，Internet的出现迅速改变着社会的面貌。国际上操作系统的研究活动也随之发生深刻的变化。

1991年，Linus Torvalds在Internet上发了一则消息，说用户可以自由下载他开发的Linux操作系统版本。逐渐地，Linux从一个人的产品，变成了通过Internet面普及的一个操作系统。Linux实际上是具有自由版权的UNIX类操作系统的一个代表。

7. 当代操作系统的两大发展方向——宏观应用与微观应用

在当代，操作系统的发展正在呈现更加迅猛的发展态势。从规模上看，操作系统向着大型和微型的两个不同的方向发展着。大型系统的典型是分布式操作系统和机群操作系统。而微型系统的典型则是嵌入式操作系统。

分布式操作系统和机群操作系统是为适应计算平台向异构、网络化演变而出现的。分布式系统是由多个连接的处理资源组成的计算系统，它们在整个系统的控制下可合作执行一个共同任务，最少依赖于集中的程序、数据或硬件。这些资源可以是物理上相邻的，也可以是在地理上分散的。机群操作系统适用于由多台计算机构成的集群。

操作系统向微型化方向发展的典型是嵌入式操作系统。在当代，嵌入式操作系统正在得到越来越广泛的应用。

1.5 操作系统分类

根据操作系统在用户界面的使用环境和功能特征的不同，操作系统一般可分为三种基本类型，即批处理系统、分时系统和实时系统。随着计算机体系结构的发展，又出现了许多种操作系统，它们是嵌入式操作系统、个人操作系统、网络操作系统和分布式操作系统。

1. 批处理操作系统

批处理（Batch Processing）操作系统的工作方式是：用户将作业交给系统操作员，系统操作员将许多用户的作业组成一批作业，之后输入到计算机中，在系统中形成一个自动转接的连续的作业流，然后启动操作系统，系统自动、依次执行每个作业。最后由操作员将作业结果交给用户。

批处理操作系统的特点是：多道和成批处理。因为用户自己不能干预自己作业的运行，一旦

发现错误不能及时改正，从而延长了软件开发时间，所以这种操作系统只适用于成熟的程序。

批处理操作系统的优点是：作业流程自动化、效率高、吞吐率高。缺点是：无交互手段、调试程序困难。

2. 分时操作系统

分时（Time Sharing）操作系统的工作方式是：一台主机连接了若干个终端，每个终端有一个用户在使用。用户交互式地向系统提出命令请求，系统接受每个用户的命令，采用时间片轮转方式处理服务请求，并通过交互方式在终端上向用户显示结果。用户根据上步结果发出下道命令。

分时操作系统将CPU的时间划分成若干个片段，称为时间片。操作系统以时间片为单位，轮流为每个终端用户服务。每个用户轮流使用一个时间片而使每个用户并不感到有别的用户存在。

分时系统具有多路性、交互性、“独占”性和及时性的特征。多路性是指，同时有多个用户使用一台计算机，宏观上看是多个人同时使用一个CPU，微观上是多个人在不同时刻轮流使用CPU。交互性是指，用户根据系统响应结果进一步提出新请求（用户直接干预每一步）。“独占”性是指，用户感觉不到计算机为其他人服务，就像整个系统为他所独占。及时性指，系统对用户提出的请求及时响应。

常见的通用操作系统是分时系统与批处理系统的结合。其原则是：分时优先，批处理在后。“前台”响应需频繁交互的作业，如终端的要求；“后台”处理时间性要求不强的作业。

3. 实时操作系统

实时操作系统（Real Time Operating System, RTOS）是指使计算机能及时响应外部事件的请求，在规定的严格时间内完成对该事件的处理，并控制所有实时设备和实时任务协调一致地工作的操作系统。实时操作系统主要追求的目标是：对外部请求在严格时间范围内做出反应，有高可靠性和完整性。

4. 嵌入式操作系统

嵌入式操作系统（Embedded Operating System）是运行在嵌入式系统环境中，对整个嵌入式系统以及它所操作、控制的各种部件装置等资源进行统一协调、调度、指挥和控制的系统软件。

5. 个人计算机操作系统

个人计算机操作系统是一种单用户多任务的操作系统。个人计算机操作系统主要供个人使用，功能强、价格便宜，可以在几乎任何地方安装使用。它能满足一般人操作、学习、游戏等方面的需求。个人计算机操作系统的主要特点是计算机在某一时间内为单个用户服务；采用图形界面人机交互的工作方式，界面友好；使用方便，用户无需专门学习，也能熟练操纵机器。

6. 网络操作系统

网络操作系统是基于计算机网络的，是在各种计算机操作系统上按网络体系结构协议标准开发的软件，包括网络管理、通信、安全、资源共享和各种网络应用。其目标是相互通信及资源共享。

7. 分布式操作系统

大量的计算机通过网络被连结在一起，可以获得极高的运算能力及广泛的数据共享。这种系统被称作分布式系统（Distributed System）。

分布式操作系统的特征是：统一性，即它是一个统一的操作系统；共享性，即所有的分布式系统中的资源是共享的；透明性，其含义是用户并不知道分布式系统是运行在多台计算机上，在用户眼里整个分布式系统像是一台计算机，对用户来讲是透明的；自治性，即处于分布式系统的多个主机都处于平等地位。

分布式系统的优点是它的分布式。分布式系统可以以较低的成本获得较高的运算性能。分布式系统的另一个优势是它的可靠性。由于有多个CPU系统，因此当一个CPU系统发生故障时，整个系统仍旧能够工作。对于高可靠的环境，如核电站等，分布式系统是有其用武之地的。

机群（cluster）是分布式系统的一种，一个机群通常由一群处理器密集构成。机群操作系统是分布式操作系统的一个新品种，有了机群操作系统，可以用低成本的微型计算机和以太网设备等产品，构造出性能相当于超级计算机运算性能的机群。

网络操作系统与分布式操作系统在概念上的主要区别是：网络操作系统可以构架于不同的操作系统之上，也就是说它可以在不同的本机操作系统上，通过网络协议实现网络资源的统一配置，在大范围内构成网络操作系统。在网络操作系统中并不要求对网络资源进行透明的访问，即需要显式地指明资源位置与类型，对本地资源和异地资源访问区别对待。

分布式比较强调单一性，它是由一种操作系统构架的。在这种操作系统中，网络的概念在应用层被淡化了。所有资源（本地的资源和异地的资源）都用同一方式管理与访问，用户不必关心资源在哪里，或者资源是怎样存储的。

1.6 研究操作系统的几种观点

在机器语言一级的计算机体系结构，大多数是很原始的。在这一级的程序设计，尤其是为I/O进行的程序设计，显得十分笨拙。一般的程序员不愿意陷入这一硬件细节的泥潭中去。相反，他们希望处理一种简单而又高级的抽象。例如，对磁盘的抽象：磁盘是一个文件卷，它有一批命名的文件，文件可以打开供存取之用，然后可以读写，最后关闭它们。这样，程序员无需关心磁盘上数据的物理位置、各磁道的区段号码、扇区之间的间隙、控制器返回的状态和错误字段、甚至驱动器的电机是否启动、启动延迟时间大小等硬件细节。

能够向程序员隐蔽硬件的真象，对可供读写的文件实行“按名存取”，并做巧妙而简捷的处理的系统自然是操作系统。另外操作系统还能隐蔽关于中断、计时和存储器管理等杂务。因此，操作系统向用户提供了一个与硬件等价，但比硬件更易于进行程序设计的扩展的机器（有时也称为虚拟机器）。

1. 软件的观点

从软件的观点来看，操作系统有其作为软件的外在特性和内在特性。

所谓外在特性是指，操作系统是一种软件，它的外部表现形式，即它的操作命令定义集和它的界面，完全确定了操作系统这个软件的使用方式。比如，操作系统的各种命令、各种系统调用及其语法定义等。我们需要从操作系统的使用界面上，即从操作系统的各种命令、系统调用及其语法定义等方面，学习和研究操作系统，只有这样才能从外部特征上把握住每一个操作系统的性能。

所谓内在特性是指，操作系统是一种软件，它具有一般软件的结构特点，然而这种软件不是一般的应用软件，它具有一般软件所不具备的特殊结构。因此，我们学习和研究操作系统时就需要研讨其结构上的特点，从而更好地把握住它的内部结构特点。比如，操作系统是直接同硬件打交道的，那么就要研究同硬件交互的软件是怎么组成的，每个组成部分的功能作用和各部分之间的关系等，换言之，即要研究其内部算法。

2. 资源管理的观点

一个计算机系统包含的硬件、软件资源可以分成以下几部分：处理器（CPU）、存储器（内存和外存或称主存和辅存）、外部设备和信息（文件）。现代的计算机系统都支持多个用户、多道作业共享，那么，而对众多的程序争夺处理器、存储器、设备和共享软件资源，如何协调这些资源，并有条不紊地进行分配呢？操作系统就是负责登记谁在使用什么样的资源，系统中还有哪些资源空闲，当前响应谁对资源的要求，以及收回哪些不再使用的资源等。操作系统要提供一些机制去协调程序间的竞争与同步，要提供一些机制对资源进行合理使用，要对资源施加保护，并采取虚拟技术来“扩充”资源等。总之操作系统是一个资源的管理者。

3. 进程的观点

这种观点把操作系统看作是由若干个可以同时独立运行的程序和一个对这些程序进行协调的核心所组成。这些同时运行的程序称为进程；每个进程都完成某一特定任务（如控制用户作业的运行、处理某个设备的输入/输出……）。而操作系统的核心则控制和协调这些进程的运行，解决进程之间的通信；它从系统各部分可以并行工作为出发点，考虑管理任务的分割和相互之间的关系，通过进程之间的通信来解决共享资源时所带来的竞争问题。通常，进程可以分为用户进程和系统进程两大类，由这两类进程在核心控制下的协调运行来完成用户的作业要求。

4. 虚机器观点

从服务用户的机器扩充的观点来看，操作系统为用户使用计算机提供了许多服务功能和良好的工作环境。用户不再直接使用硬件机器（称为裸机），而是通过操作系统来控制和使用计算机，从而把计算机扩充为功能更强、使用更加方便的计算机系统（称为虚拟计算机）。操作系统的全部功能，如系统调用、命令、作业控制语言等，被称为操作系统虚机器。

虚机器观点从功能分解的角度出发，考虑操作系统的结构，将操作系统分成若干个层次，每一层次完成特写的功能，从而构成一个虚机器，并为上一层次提供支持，构成它的运行环境。通过逐个层次的功能扩充，最终完成操作系统虚机器，从而向用户提供全套的服务，完成用户的作业要求。

5. 服务提供者观点

在操作系统以外，从用户角度看操作系统，则它应能为用户提供比裸机功能更强、服务质量更高、使用户更觉方便灵活的虚拟机器。操作系统能为用户提供一组功能强大的、方便、好用的广义指令（系统调用）。

1.7 Windows操作系统的发展历程

从1983年微软公司宣布Windows的诞生到现在的Windows XP，Windows已经走过了将近18

年的历史。本节介绍Windows操作系统的发展历程以及主要版本的功能特性。

1.7.1 Windows的开发过程

Windows的起源可以追溯到美国Xerox公司进行的工作。该公司著名的研究机构Palo Alto Research Center (PARC), 于1981年宣布推出世界上第一个商用的GUI (图形用户接口) 系统: Star 8010工作站。

当时, Apple Computer公司的创始人之一Steve Jobs, 在参观Xerox公司的PARC研究中心后, 认识到了图形用户接口的重要性以及广阔的市场前景, 开始着手进行自己的GUI系统研究开发工作, 并于1983年研制成功第一个GUI系统: Apple Lisa。随后不久, Apple又推出第二个GUI系统Apple Macintosh, 这是世界上第一个成功的商用GUI系统。

图形界面的优势, 人人可见, 这是未来趋势, 早在1981年, 微软公司内部就制定了发展“界面管理者”的计划。到了1983年5月, 微软公司决定把这一计划命名为Microsoft Windows。

1983年11月10日, 比尔·盖茨宣布推出Windows, 但是一直到1985年11月微软公司才正式发布Windows 1.0版。Windows这个产品在微软公司的历史上创造了几个记录: 延迟交货次数最多, 投入开发人员最多, 开发时间最长, 更换主管人员最多。几年之后, Windows也创造了销售成绩最佳的历史记录。

1987年12月, Windows 2.0正式供货。1990年5月22日, 微软推出Windows 3.0。该版本的Windows的许多功能都比以前大有提高。从此, 在许多独立软件开发商和硬件厂商的支持下, 微软Windows在市场中逐渐开始取代DOS成为操作系统平台的主流软件。

1.7.2 Windows的版本

本节列出Windows的版本发展时间 (见表1-1)。

表 1-1

Windows 9X内核系列的发展	Windows NT内核系列的发展
1983年11月: Windows宣布诞生	
1985年11月: Windows 1.0	
1987年4月: Windows 2.0	
1990年5月: Windows 3.0	
1992年4月: Windows 3.1	
	1993年8月: Windows NT 3.1
1994年2月: Windows 3.11	1994年9月: Windows NT 3.5
1995年8月: Windows 95	1995年6月: Windows NT 3.51
	1996年8月: Windows NT 4.0
	1997年9月: Windows NT 5.0 Beta 1
1998年6月: Windows 98	1998年8月: Windows NT 5.0 Beta 2
1999年5月: Windows 98 SE	1999年4月: Windows 2000 Beta 3
1999年11月: Windows Millennium Edition Beta 2	

(续)

Windows 9X内核系列的发展	Windows NT内核系列的发展
2000年9月：Windows Me	2000年2月：Windows 2000 2000年7月：Windows 2000 SP1, Windows Whistler Developer Preview 2000年10月：Windows Whistler Beta 1 2001年3月：Windows XP Beta 2
2001年1月：Windows 9x内核正式宣告停止	

1.7.3 Windows 早期版本的技术特点

1. Windows 1.0和Windows2.0

微软公司在1985年和1987年分别推出的Windows 1.0版和Windows 2.0版是基于Intel x86微处理芯片计算机上的操作系统。但是，由于当时硬件和DOS操作系统的限制，这两个版本并没有取得很大的成功。

2. Windows 3.0

微软公司于1990年5月推出Windows 3.0版。该版本对内存管理、图形界面做了重大改进，使图形界面更加美观，并支持虚拟内存。这个“千呼万唤始出来”的操作系统一经面世便在商业上取得惊人的成功：不到6周，微软销出50万份Windows 3.0拷贝，打破了任何软件产品的6周销售记录，从而一举奠定了微软公司在个人计算机操作系统上的垄断地位。

3. Windows 3.1

微软公司推出的Windows 3.1版对Windows 3.0版做了一些改进，引入了可缩放的TrueType字体技术，还引入了一种新的文件管理程序，改进了系统的可靠性。更重要的是，微软公司增加了对对象链接和嵌入技术（OLE）以及对多媒体技术的支持。

Windows 3.0和Windows 3.1都必须运行于MS-DOS操作系统之上。

4. 早期Windows的成功之处

Windows之所以取得成功，主要在于它具有以下优点：

1) 直观、高效的面向对象的图形用户界面，易学易用。

Windows用户界面和开发环境都是面向对象的。用户采用“选择对象-操作对象”这种方式进行工作。比如要打开一个文档，首先用鼠标或键盘选择该文档图标，然后打开该文档。这种操作方式模拟了现实世界的行为，易于理解、学习和使用。

2) 用户界面统一、友好、美观。

Windows应用软件大多拥有相同的或相似的基本外观，包括窗口、菜单、工具条等。用户只要掌握其中一个软件，就不难学会其他软件，从而降低了用户学习掌握有关软件的门槛。

3) 丰富的设备无关的图形操作。

Windows的图形设备接口（ODI）提供了丰富的图形操作函数，并支持各种输出设备。设备无关的特性意味着在针式打印机上和高分辨率的显示器上都能显示出相同效果的图形。

4) 多任务。

Windows是一个多任务的操作环境，它允许用户同时运行多个应用程序，或在一个程序中同时做几件事情。每个程序在屏幕上占据一块矩形区域，称为窗口。用户可以移动这些窗口，或在不同的应用程序之间进行切换，并可以在程序之间进行手工的和自动的数据交换和通信。

5) 丰富的Windows软件开发工具。

随着Windows的普及，各软件公司纷纷推出新一代可视化开发工具，如Visual Basic、Visual C++、Borland C++ Builder、Delphi和用于数据库开发的Power Builder、Visual FoxPro等。其中Visual C++目前已成为应用最广泛的高级程序设计语言之一。

6) 面向对象式的程序设计思想。

在Windows的界面设计和软件开发环境中，处处贯穿着面向对象的思想。在Windows中，Windows程序的执行过程本身就是窗口和其他对象的创建、处理和消亡过程。因此，用面向对象方法来进行Windows程序的设计与开发是极其方便的和自然的。

1.7.4 Windows 95和Windows 98

1. Windows 95

微软公司于1995年推出Windows 95（又名Chicago），它可以独立运行而无需DOS支持。

Windows 95是Windows操作系统发展史上一个重要的作品。Windows 95采用32位处理技术，兼容以前16位的应用程序，在Windows发展史上起到了承前启后的作用。

Windows 95对Windows 3.1版做了许多重大改进。这些改进包括：更加优秀的、面向对象的图形用户界面，全32位的抢先式多任务和多线程，内置的对Internet的支持，更加高级的多媒体支持（声音、图形、影像等），即插即用，32位线性寻址的内存管理和良好的向下兼容性等。

Windows 95其实是16位和32位混合在一起的操作系统，这种操作系统最大的问题就是稳定性较差。

2. Windows 98

1998年6月，微软公司发布Windows 98。Windows 98仍兼容16位的应用程序，是Windows系列产品中最后一个“照顾”16位应用程序的操作系统。

Windows 98有许多Windows 95所不具备的新的特点，下面给予简要的介绍。

(1) Internet Aware

1) Web-Aware 用户界面。使用Web-Aware用户界面，因特网成为用户界面的一部分。

2) 高级的因特网浏览功能。Windows 98提供了容易、迅捷地浏览网络的方法。支持主要的因特网标准，包括：HTML、Java、ActiveX、Java Script、Visual Basic Scripting，以及主要的安全标准。提供动态HTML、just-in-time Java编译器等。

3) 个性化的因特网信息发布。Windows 98为在线通信提供了丰富的工具，包括：OutLook Express、Microsoft Net Meeting等。

4) 拨号网络的改进。拨号网络已经更新。在拨号连入因特网或公司网络时，会产生显著的性能改善。

(2) FAT 32

FAT 32是FAT文件系统的一个改进版本，它允许把超过2G的硬盘格式化为一个单一驱动器，这使大磁盘上的空间得到更有效的利用，用户平均多得28%的硬盘空间。

(3) 电源管理的改进

Windows 98提供内置的对先进配置与电源接口 (Advanced Configuration and Power Interface, ACPI) 的支持。

(4) Win32驱动程序模型 (Win32 Driver Model, WDM)

Win32驱动程序模型是一个对Windows 95和Windows NT全新的、统一的驱动程序模型。WDM使得新设备对于两种操作系统有单一的驱动程序。这允许Windows 98在增加对新的WDM驱动程序的支持的同时，也保持完整的对传统设备驱动程序的支持。

(5) 多种加强功能

比较重要的加强功能有：

1) Microsoft系统信息工具4.1。这个工具由一系列ActiveX控件组成，每一个控件负责收集并在Microsoft System Information Utility的恰当位置中显示一个特定种类的系统信息。

2)注册表检查专家 (Registry Checker)。注册表检查专家是一个发现并解决注册表问题，定期备份注册表的程序。

3) 自动忽略驱动程序代理 (Automatic Skip Driver Agent, ASDA)。自动忽略驱动程序代理识别出那些已知的会造成Windows 98停止响应的、潜在危险的故障，标记它们，以便在随后的开机中忽略它们。

4) 系统配置工具 (System Configuration Utility)。图形化系统配置工具允许用户通过使用复选框来解决问题，允许用户创建和恢复备份配置文件。

5) 分布式的部件对象模型 (Distributed Component Object Model, DCOM)。部件对象模型允许软件开发者创建部件应用程序。

6) Active Movie。Active Movie是一种针对Windows的新的媒体传输体系，它在提供高品质的视频播放的同时，还展示了一组用于建立多媒体应用程序与工具的接口。Active Movie支持流行的媒体类型的播放，包括：MPEG音频、WAV音频、MPEG视频、AVI视频和Apple Quick Time视频。

7) 对新一代硬件的支持 (Support for New Generation of Hardware)。Windows 98的一个主要目的就是为用户可以使用一批最近几年在计算机硬件方面的创新提供完全的支持，包括：Universal Serial BUS (USB)、IEEE 1394、Accelerated Graphics Port (AGP)、Advanced Configuration and Power Interface (ACPI) 和DVD。

8) 实现计算机和辅助设备的强大功能。Windows 98操作系统提供了对外围设备的内置支持，不但支持普通外围设备，而且还支持新一代的外围设备，例如操纵杆、游戏面板、数字相机、扫描仪、声卡、电视调谐卡等。

1.7.5 Windows NT操作系统的技术特点

本节简要的介绍Windows NT操作系统的技术特点。

1. Windows NT的设计

Windows NT开发小组于1989年成立的时候，任务十分明确：开发设计一种个人计算机操作系统，满足现在和将来PC平台上计算机操作系统的需要。其设计目标是：

(1) 鲁棒性

操作系统必须主动地保护自身免受内部异常和外部有意或无意破坏的影响，并且必须对软件和硬件的错误做出可预测的响应。系统的结构和编码实现必须直截了当，接口和行为描述必须规范。

(2) 可扩展性和可维护性

Windows NT的开发必须面向未来。Windows NT的升级应该能够满足初始设备制造厂家（OEM）和微软公司的未来需求。NT系统必须具有可维护性，即对于NT支持的应用程序接口（API）集，NT必须能适应其改变和增加，而不是要求API使用标志或其他设备来剧烈改变它们的功能。

(3) 可移植性

系统只需做很小的再编码就可工作于不同的计算机平台。

(4) 高性能

为获得高性能并进而得到系统的灵活性，在系统的设计中必须采用一些好的算法和数据结构。

(5) 兼容POSIX并满足美国政府的C2安全标准

POSIX标准要求操作系统供应商采用UNIX风格的接口，这样应用程序就易于从一个系统搬到另一个系统。美国政府的安全规定要求操作系统具有一定的安全保护措施，诸如帐号检查、系统接入检测、各用户资源分配和资源保护等。系统设计中包含这些特性后，Windows NT就可应用于政府部门。

2. Windows NT系统功能

整个Windows NT系统的设计包括一个功能强大的执行模块，它运行于特权（或核心）处理器模式下。系统设计还提供系统服务、内部处理和一套称为受保护的子系统的非特权服务器。这些子系统运行于执行模块外的非特权（或用户）模式下。值得注意的是，执行模块提供进入系统的唯一入口，任何其他损坏安全或破坏系统的可能入口都是不存在的。

一个受保护的子系统可以作为一个常规（本地）进程运行于用户模式下。与应用程序相比，子系统也可以有一些扩展的权力，但是它不能看成是执行模块的一部分。因此，子系统不能越过系统安全结构或使用其他方式对系统造成破坏。子系统使用高性能的本地过程调用（Local Procedure Call, LPC）与它们的客户机进行通信，或互相之间进行通信。

NT执行模块包括一套用于系统服务的组成部分：对象管理器（Object Manager）、系统安全监控器（Security Reference Monitor）和进程管理器（Process Manager）等。这些模块的主要功能是从发出请求的子系统或应用程序中选定一个已经存在的线程（thread）。首先它判断要处理的线程是否有效，然后执行这个线程并把线程的控制权交回发出请求的程序。

(1) 可维护性和可扩展性

为满足Windows NT可维护性和可扩展性的要求，采取了以下措施：

1) 将系统设计得十分简洁,并提供可扩展的编程文档。在整个系统设计中都使用了通用的编程标准,程序编码就像文档一样直截了当,使得后续的编程开发人员能够完成系统设计中的任何一块小的工作。

2) 由于使用子系统来实现系统的主要部分,因此Windows NT能隔离并控制所依赖的系统环境。例如,POSIX标准的变化只会影响一个系统组成部分,即POSIX子系统,进程结构的设计、内存管理和同步原语等都不会受到影响。

3) Windows NT设计适应了需求的改变和增长。子系统可以在不对基本系统产生影响的情况下增加系统的功能。可以在不修改Windows NT执行模块的情况下,加入新的子系统。

4) 最重要的是所有子系统经过编码实现后,都能利用Windows NT的安全特性。

5) 在Windows NT 4.0里,许多Win32的图形用户界面(GUI)子系统,如窗口管理器(Window Manager)、图形设备界面(GDI)和相关的图形驱动程序等,都从运行于csrss.exe子系统进程里的一段代码移到核心模式设备驱动程序(win32k.sys)。控制台、系统关闭和硬件错误处理等部分仍然保留在用户模式下。这种改变大大提高了系统性能,同时降低了内存需要,对应用程序开发人员没有丝毫影响。

(2) 内置鲁棒性

Windows NT通过如下几点达到鲁棒性的设计目标。

1) 系统的核心模式部分输出定义精确的API,通常没有模式参数或其他不可思议的标志。因此,API实现简单、测试容易和归档方便。

2) 系统主要组成部分(如Win32、OS/2和POSIX)都被分割成独立的子系统,使每个子系统设计简单良好。每个子系统要实现的只是其API集合需要的某些特性。

3) 在设计中广泛使用基于帧的异常控制器(异常控制器与一段特定子程序或某个子程序的一部分相联系),这使得Windows NT和其子系统能以一种可靠有效的方式捕捉编程错误、滤除坏的或无法寻址的参量。

4) 由于操作系统划分成核心模式系统服务和子系统,因此系统通过参量有效性的判断能更加有效地防止运行不良的应用程序破坏操作系统。

(3) Windows NT 参量的有效性

为了使Windows NT达到鲁棒性的目标,必须保证:不可能通过传递一个无效的参量值、传递一个调用者不能修改的指向内存的指针、或在执行线程的同时剧烈改变或删除参量占用的内存的方法来造成系统崩溃或对系统产生损害。

1.7.6 Windows Embedded家族

本节向读者简短地介绍Windows Embedded家族的产品系列。

Windows Embedded操作系统产品家族

Microsoft Windows Embedded操作系统产品家族由三种操作系统组成。

(1) Windows CE 3.0

Windows CE 3.0是一种针对小容量、移动式、智能化、32位、连接设备的模块化实时嵌入式

操作系统。

(2) Windows NT Embedded 4.0

Windows NT Embedded 4.0是一种针对基于PC体系结构解决方案的全功能嵌入式操作系统。Windows NT Embedded 4.0操作系统采用PC体系结构，并继承了Windows NT 4.0的全部服务与功能，可用于快速建立功能强的嵌入式系统。

(3) 带有Server Appliance Kit的Windows 2000

使用带有Server Appliance Kit的Windows 2000，可以快捷地建立具有Windows 2000功能的专用服务器。

1.7.7 Windows 2000

本节介绍Windows 2000操作系统的简要功能特点。

1. 产品系列

从笔记本电脑到高端服务器，Windows 2000 平台是下一代PC的商务操作系统。该平台建立于NT技术之上，具有强可靠性，高可用时间，它通过简化系统管理降低了操作耗费，是一种适合从最小的移动设备到最大的电子商务服务器新硬件的操作系统。

Windows 2000 系列包括以下产品：

- Windows 2000 Professional。
- Windows 2000 Server。
- Windows 2000 Advanced Server。
- Windows 2000 Datacenter Server。

2. Windows 2000 Professional介绍

Windows 2000 Professional是Windows NT Workstation新版本的新名称。

Windows 2000 Professional继承了Windows NT的技术，提供了高层次的安全性、稳定性和系统性能。同时，它帮助用户更加容易地使用计算机、安装和配置系统以及浏览Internet等。而对于系统管理员而言，Windows 2000 Professional 是一套更具有可管理性的桌面系统。

Windows 2000 Professional具有以下特点：

- 1) 友好的Windows，使用、安装、配置系统和浏览Internet都很容易。
- 2) 以Windows NT Workstation的技术为基础，采用标准化的安全技术，具有工业级的可靠性和更高的性能。
- 3) 继承了Windows 98特性，让移动用户也能够方便地工作，并且广泛支持新一代的硬件设备。
- 4) 更具有可管理性，这意味着更低的总体拥有成本。

2. Windows 2000 Server系列介绍

Windows 2000 Server这个版本以前的名称是Windows NT Server 5.0，它是在Windows NT Server 4.0的基础上开发出来的。

Windows 2000 Server是为服务器开发的多用途操作系统，可为部门工作小组或中小型公司用户提供文件和打印、应用软件、Web和通信等各种服务。它是一个性能更好、工作更稳定、更容

易管理的平台。

Windows 2000 Server 最重要的改进是，在“活动目录”目录服务技术的基础上建立了一套全面的、分布式的底层服务。它能有效地简化网络用户及资源的管理，并使用户更容易地找到企业网为他们提供的资源。

Windows 2000 Server支持两路对称多处理器（SMP）系统，是适用于中小型企业应用程序的开发、Web服务器、工作组和分支部门的操作系统。

有几种版本的Windows 2000 Server。其中，Windows 2000 Server用于工作组和部门服务器；Windows 2000 Advanced Server用于应用程序服务器和更强劲的部门服务器；Windows 2000 Datacenter Server用于运行核心业务的数据中心服务器系统。

(1) Windows 2000 Server

Windows 2000 Server特点如下：

1) 全面的Internet和应用软件服务。通过Internet 服务集成，Windows 2000 Server系列使建立并部署电子商务、知识管理和其他商业方式更为容易。

2) 增强的可靠性和可扩展性。与Windows NT 4.0相比，Windows 2000 Server具有更高水平的整体系统可靠性和规模性。例如，系统已针对32 位处理器进行了优化，支持高达64GB的内存，并建立了更强大的系统体系。

3) 强大的端对端管理使成本更低。为降低成本，Windows 2000 Server 为服务器、网络 and 基于 Windows 的客户系统提供综合的管理服务。

(2) Windows 2000 Advanced Server

这个版本以前的名称是 Windows NT Server 5.0 Enterprise Edition。

Windows 2000 Advanced Server除了具有Windows 2000 Server的所有功能和特性之外，还有一些专为大型的企业级服务器所设计的特性，例如群集、加载平衡和对称多处理器（SMP）支持等。它能够为客户提供一个高可靠性和高扩展性的平台，可承担起运行企业核心业务软件的重任，包括数据库、记录和通告、联机交易处理和企业资源管理（ERP）系统等。

Windows 2000 Advanced Server具有以下特性和功能：

1) 更强的SMP扩展能力。提供更强的对称多处理器支持，支持数可达四路。

2) 群集功能。Windows 2000 Advanced Server的群集功能具有更强大的群集功能。

- 更高的稳定性。可为核心业务提供更高的稳定性，在多种一般错误发生后一分钟内自动重启应用软件。

- 网络负载平衡。为网络服务和应用程序提供高可用性和扩展能力。

- 组件负载平衡。为COM+组件提供高可用性和扩展能力。

3) 高性能排序。Windows 2000 Advanced Server优化了大型数据集的排序功能。

这些功能和特性使Windows 2000 Advanced Server比Windows 2000 Server具有更高的扩展性、互操作性和可管理性，可用于拥有多种操作系统和提供Internet服务的部门和应用程序服务器。

(3) Windows 2000 Datacenter Server

Windows 2000 Datacenter Server版本，是Microsoft提供的功能最强的服务器操作系统。

Windows 2000 Datacenter Server支持16路对称多处理器系统以及高达64GB的物理内存。它将群集和负载平衡服务作为标准的特性。另外，它为大型的数据仓库、经济分析、科学和工程模拟、联机交易服务等应用进行了专门的优化。

4. Windows 2000 平台

Windows 2000 Professional和Windows 2000 Server结合起来构成Microsoft Windows 2000平台，形成先进的、基于PC的客户/服务器平台，能够降低总体拥有成本、提供可靠的7 x 24计算。

1.7.8 Windows XP

Windows XP是一个把消费型操作系统和商业型操作系统融合为统一系统代码的Windows，它结束了Windows两条腿走路的历史，所以它也是第一个既适合家庭用户，也适合商业用户使用的新型Windows。

截止到2001年6月，微软公司还没有正式公布Windows XP。下面介绍Windows XP Home Edition和Windows XP Professional具有的新特性，均基于测试版本。

1. Windows XP Home Edition的新特性

Windows XP Home Edition是一个易于使用的智能化家用操作系统，其特点有：

- 1) 更丰富的通信功能。即时语音、视频和应用程序共享功能可使用户之间的通信交流更为高效。
- 2) 更高的可移动性。笔记本用户可以随时随地访问他们的信息。
- 3) 改进的帮助与支持。用户在遇到困难时能够与其他用户或帮助资源相连接，从而及时获得帮助与支持。
- 4) 简洁的数码影像。Windows XP将会使创建、管理、共享数码影像变得非常轻松。
- 5) 令人激动的音乐和娱乐。Windows XP为发现、下载、个性化、播放高品质的音频和视频内容提供良好的支持。
- 6) 提高家庭两络品质。Windows XP使用户能够轻松地与家人共享信息、设备、两络连接。

2. Windows XP Professional

本节，我们简要地介绍Windows XP Professional版的特点。

(1) 运行新特性

1) 基于新型Windows引擎。Windows XP Professional建立在Windows NT和Windows 2000代码基础之上，采用了32位计算体系结构和完全受保护的内存模型。这使得Windows XP Professional成为可靠的操作系统。

2) 系统还原。系统还原特性自动地创建简单的可标识还原点，可以让用户和管理员在不丢失数据的前提下，将计算机还原到以前的状态。系统还原功能不恢复用户的数据或文档文件，因此还原工作不会丢失用户的数据、电子邮件等。

3) 设备驱动程序回滚。当安装了特定类型的新设备驱动程序时，Windows XP Professional将备份以前安装的驱动程序信息，这样如果新的设备驱动程序引起了Windows XP Professional故障，管理员可以轻松地重新安装以前使用的驱动程序。

4) 增强的设备驱动程序检验器。Windows XP Professional使用Windows 2000的设备驱动程序检验器,可以给设备驱动程序提供功能更强的负载测试。经过测试的设备驱动程序将会是健壮的驱动程序,它可以保证系统最大的稳定性。

5) 减少重启的情况。Windows XP Professional消除了大部分需要最终用户重新启动计算机的情况,用户可以体验到更高级别的系统运行时间。

6) 改良的代码保护。重要的内核数据结构都是只读的,因此驱动程序和应用程序都不会破坏它们。所有的设备驱动程序代码都是只读的,并且是页保护的。恶意的应用程序将不能影响核心操作系统区域。

7) 可伸缩内存和处理器。最大可以支持4GB RAM和两个对称多处理器。

(2) 防止应用程序错误的手段

1) 并行DLL支持。提供安装多个不同Windows组件版本的机制,并且可以并行运行。这可以让使用一种系统组件版本编写和测试的应用程序继续运行,这样就可以解决“DLL hell”(DLL魔窟)问题。

2) Windows文件保护。保护核心代码不被安装的应用程序覆盖。通过保护系统文件,Windows XP Professional预防了早期Windows版本中最常见的系统失败错误。

3) Windows安装程序。可以帮助用户正确地安装、配置、跟踪、升级和卸载软件程序。可以减小系统的故障时间,提高系统的可靠性。

4) 增强的防病毒功能。为了更好地防止电子邮件病毒攻击,Windows XP Professional缺省情况下不允许执行电子邮件附件中的程序。当然系统管理员可以远程管理(通过组策略)系统,而这个时候就可以允许执行特定的文件类型或应用程序。管理员在保护系统免受电子邮件病毒攻击时,有更高一级的控制权力。

(3) 增强Windows安全性

1) Internet连接防火墙。防火墙客户端可以保护用户不受一般的Internet攻击。

2) 带有多用户支持的加密文件系统(简称EFS)。可以使用任意产生的密钥加密文件。加密和解密过程对用户来说是透明的。在Windows XP Professional中,EFS可以让多个用户访问加密的文档。这是保护不受黑客和数据盗窃的最高级别。

3) IP安全(IP Sec)。IP Sec是给虚拟专用网(VPN)提供安全性的重要组成部分,它可以让企业在Internet上安全地传输数据。系统管理员可以快速简便地构建虚拟专用网。

4) Kerberos支持。Kerberos是一个Internet标准,适用于包括不同网络的操作系统(例如UNIX)。Windows XP Professional可以给Windows 2000和“Whistler”服务器以及UNIX平台的最终用户提供验证。

5) 智能卡支持。智能卡性能集成到操作系统中,支持智能卡登录到终端服务器会话。智能卡增强了软件验证的解决方案,例如客户端验证、交互登录、代码签名和安全电子邮件。

(4) 简化的管理和部署

1) 增强的应用程序兼容性。数百万个不能运行在Windows 2000 Professional的应用程序将可以运行在Windows XP Professional上。

2) 用户状态移植工具。用户状态移植工具帮助管理员将用户的数据和应用程序/操作系统设置从旧的计算机中移植到新的Windows XP Professional桌面计算机中。在移植后,可以减少IT管理员的恢复工作量,并且还可以减少最终用户的停机时间,因为他们可以继续使用熟悉的操作环境。

3) 系统准备工具 (SysPrep)。SysPrep可以帮助管理员备份计算机配置、系统和应用程序信息。包括操作系统和商务应用程序的系统映像可以用于多个不同的计算机配置。SysPrep可以让管理员减少需要维护的操作系统映像数,并且降低配置一般的桌面系统所需要的时间。

4) 组策略。组策略设置可以让管理员以逻辑单元(例如部门或办公地点)的形式组织用户和对象,然后给这些逻辑单元分配相同的设置,包括安全、外观和管理选项,这个过程可以简化相应的管理任务。除了Windows 2000 Professional提供的策略以外,Windows XP Professional还提供了数百个新的策略。尽管用户不停地改变办公地点,但是他们仍然可以访问重要的数据,可以维护自己定制的工作环境。

5) RsoP。通过使用RsoP,管理员就可以有一个强大的、灵活的工具来规划、预览、监视和调试组策略。

6) Microsoft管理控制台(MMC)。MMC为管理工具提供了一个集中管理的、一致的环境。IT管理员现在可以创建定制的应用程序控制台。

7) 恢复控制台。这是一个命令行式的控制台,通过这个控制台可以启动和终止服务、格式化硬盘、从本地硬盘读取数据、将数据写回本地硬盘以及执行很多其他的管理任务。

8) Windows管理规范(WMI)。WMI为监视和管理系统资源提供标准的基础构架。可以让系统管理员通过编写脚本和使用第三方应用程序来监视和控制系统。

9) 安全模式启动选项。可以让Windows XP Professional以最基本的方式启动系统,使用缺省的设置和最小的设备驱动程序。提供一种将系统启动到GUI的方法,这样IT专业人士可以在这种方式下修复操作系统。

10) 崭新的可视化设计。在保留使用Windows 2000核心技术的同时,Windows XP Professional还提供了一个崭新的可视化设计。在这个操作系统中,统一并且简化了一般的任务,添加了新的可视化界面帮助用户使用计算机。管理员或者最终用户可以选择使用这个更新的用户界面或者选择使用典型的Windows 2000单击按钮界面。以更简单的方式提供许多一般的任务,以使用户充分利用Windows XP Professional。

(5) 革新远程用户工作方式

1) 远程桌面。可以让用户通过任何计算机和网络连接访问他的计算机及它上面的任何程序和数据。通过使用Microsoft的远程桌面协议(RDP),用户可以用低功率计算机通过任何网络连接访问桌面计算机的所有数据和应用程序。

2) 证书管理器。安全地保存口令信息。可以让用户在第一次输入用户名和口令后,以后由系统自动提供。如果用户没有连接到域,或在没有信任关系的情况下希望访问多个域的资源,通过该功能可以轻松地访问网络资源。

3) 脱机文件和文件夹。当用户断开网络连接时,可以指定需要脱机阅览哪个基于网络的文件

和文件夹。现在可以加密脱机文件夹提供最高级别的安全性。用户在断开网络连接之后，可以使用同样的方法操纵文档。

4) 同步管理器。可以让用户将他们的脱机文件和文件夹与网络上的相应内容进行比较并更新。如果脱机使用文件和文件夹，此功能可以自动将更新信息复制到网络，以确保网络上的信息是最新的。

1.7.9 Windows 2000开发的艰辛与规模

Windows 2000是迄今为止（在Windows XP 尚未推出之时），微软公司历史上最艰巨的开发任务。了解以下Windows 2000开发过程的片段是有意义的。

1. 对话

这里记录了微软公司Windows业务部高级副总裁Brian Valentine 和项目经理Iain McDonald与记者的部分对话。

记者：有多少人参与了Windows 2000的开发？

Valentine：我们核心部门的成员有2500人。不过，微软公司的每一位员工都为Windows 2000做出了贡献。

记者：核心开发部的成员多长时间开一次会？

McDonald：我们有一个会议室，我们称之为“作战室”，这就是项目管理中心。项目的关键人员经常在这里会面，通常是每周7天。根据项目的进展情况有时一天的会晤达3次之多。我们之所以开这么多会议，往往是为了集中兵力研究新出现的重大课题。这样，我们就能够审查我们所取得的进展及所面临的挑战，并让每位成员及时掌握项目现状。

记者：Windows 2000中有哪些因素确保其质量？

Valentine：我们采用几项措施来确保质量。我们观察了该系统的实际运行状态。为此，我们密切观察了公司内部100台服务器的工作状况。我们在公司内部及客户的系统中进行测试，以便发现任何可能出现的错误，并且仔细地跟踪运行状态。我们还进行了强化试验，每个晚上在2000台电脑上运行，模拟二至三年的电脑使用量。然后，我们设法解决所出现的每个问题。我们非常重视一切质量问题，比以往任何时候都重视。

2. 数据

Windows 2000开发过程中产生了一些有趣的数据，列在下面供读者参考。

1) 在硬件支持方面所取得的进步：

- 支持的OEM系统的数量：580。
- 支持的打印机的数量：2000。
- 支持的网络设备的数量：700。
- 支持的调制解调器的数量：4200。
- 支持的扫描器：55。
- 支持的相机：41。

2) 多语言支持功能：

- 全球单一的二进制。
 - 完全本地化版本：23个。
 - 所支持的国际区域设置：134种。
 - 所配置的所有语言的字体数量：100MB。
- 3) 测试工作量：
- 测试用代码行数：超过1000万行。
 - 测试兼容性的应用软件数量：1000种。
 - 测试中所使用的脚本程序：6500种。
 - 每月备份的数据：88TB。
 - 每晚模拟打印数量：25万页。
 - 测试实验室占地面积：20万平方英尺。
 - 每周“烧”CD：12000盘。
- 4) 网络测试工作量：
- 所处理的DNS查询：8亿次
 - 所处理的WINS查询：2100万次。
 - 所处理的DHCP租赁：1200万个。
 - 所处理的连接管理器请求：超过10亿。
 - 所处理的SNMP PDU：650万个。
 - 采用H323和IP多播举行的远程会议：在145天内举行了11 500次。
- 5) 目录与安全测试工作量：
- 所测试的对象：10M/DC。
 - 目录数据库规模：37GB。
 - 输出的LDAP对象数量：100万个。
 - 单域内所测试的DC数量：130个。
 - Kerberos验证率：400/秒。
 - 目录内的证书数量：536 000个。
- 6) 外协单位：
- 参与Windows 2000快速部署计划（RDP）的公司数量：23家。
 - 89家OEM公司制造了333种“Windows 2000 Ready PCs”。
 - 加入首波计划（First Wave Program）的应用程序数量：300个。
 - 微软公司在客户培训方面的投资数额：4000万美元。

习题

- 1.1 什么是计算机系统？计算机系统是怎样构成的？
- 1.2 了解一个计算机系统的组成，说明其：① 硬件组织的基本结构,画出硬件配置图；② 主要系统软件和应用软件（若有的话）及它们的作用。

- 1.3 什么是操作系统？请举例说明它在计算机系统中的重要地位。
- 1.4 操作系统要做哪些事？请用-一个实际的例子来说明，比如运行“Hello World!”程序。
- 1.5 为什么说“操作系统是控制硬件的软件”的说法不确切？
- 1.6 操作系统的基本特征是什么？说明它们之间的关系。
- 1.7 试从独立性、并发性、交互性和实时性等方面比较批处理系统、分时系统和实时系统。
- 1.8 引入多道程序设计技术的起因和目的是什么？多道程序环境下需要什么样的硬件支持？多道程序系统的特征是什么？
- 1.9 多道程序设计的度是指在任一给定时刻，单个CPU所能支持的进程数目的最大值。讨论要确定一个特定系统的多道程序设计的度必须考虑的因素。可以假定批处理系统中进程数量与作业数量相同。（这些因素中的某些因素在后面章节中会有详细论述。）
- 1.10 描述批处理系统响应一个执行请求需要的时间（称为响应时间）。描述分时系统下的响应时间。什么类型的系统可能有较短的响应时间？为什么？
- 1.11 什么情况下批处理系统是比较好的策略？什么情况下分时系统是比较好的策略？
- 1.12 现有以下应用，请选择合适的操作系统：① 航空航天、核研究；② 统计局数据中心；③ 学生上机学习编程；④ 高炉炉温控制；⑤ 民航订票系统；⑥ 发送电子邮件。
- 1.13 CPU响应中断时,为什么要交换程序状态字？怎样交换？
- 1.14 有的处理器会在中断发生的时候自动将程序计数器和程序状态字转存到系统堆栈上，请说明这种实现会带来什么好处，同时也会带来哪些不好的地方。
- 1.15 外壳程序（Shell）是不是操作系统的一部分，为什么？
- 1.16 现代操作系统的设计很讲求机制与策略的分离，以使操作系统的结构和实现能够在一定范围内适应不同应用的需要。例如Solaris的调度器实现了进程调度的基本机制，同时它允许通过核心参数的动态调整实现不同负载下的系统性能平衡，这就是一种机制与策略的分离。请给出一个例子，说明怎样根据调度将机制与策略分开。请构造一种机制，允许父进程控制子进程的调度策略。
- 1.17 选择一个在本章中没有讨论到的现代操作系统，写一篇文章概述该系统如何进行设备管理、文件管理、进程管理和内存管理。不必给出对该操作系统本身的严格分析。本题的目的之一是让读者去查阅技术文献（不要拿另一本教科书作为主要信息来源）。



第 ② 章

Windows 2000/XP 的 体系结构

第 ② 章

Windows 2000/XP的体系结构

Windows 2000/XP经过十多年的发展已经从最初功能十分有限的32位操作系统演化成了现在的界面友好、管理方便、功能强大的操作系统家族，它之所以有这样大的发展，除了微软公司强大的商业推动力量之外，Windows 2000/XP本身的技术因素也是重要的原因，尤其是它自身的体系结构所具有的可扩充性、可执行性、鲁棒性、兼容性和高效性，正是这些特性让Windows 2000/XP不断得以进步。

本章将从操作系统设计的角度详细分析一下Windows 2000/XP所具有的基本体系结构和运行机理。我们将首先解决有关操作系统设计的理论问题，然后再对Windows 2000/XP所具有的特殊体系结构进行一些必要的介绍和分析。

2.1 操作系统的设计

设计操作系统并不像解一道数学习题那样有着既定的证明思路和规范化的推导过程，这是一项系统工程，有着十分复杂的过程。操作系统作为一个软件系统来说是以庞大而复杂著称的。以IBM公司的OS/360系统为例，它由4000个模块组成，共约100万条指令，花费5000人年，经费达数亿美元。但每个版本都仍然隐藏着无数的错误。其负责人Brooks在描述OS/360研制过程中的困难和混乱时曾经说过：“……巨兽在泥潭中做垂死挣扎，挣扎得越猛，泥浆就沾得越多。最后没有一个野兽能逃脱淹没在泥潭中的命运，……程序设计就像是这样一个泥潭。……一批批程序员在泥潭中挣扎……没有人料到问题会这样棘手”。这就是在20世纪60年代出现的软件危机现象，OS/360研制开发所陷入的困境，推动了软件工程方法与技术的诞生。毫无疑问，今天操作系统的开发应遵循软件工程的原则和方法。

2.1.1 操作系统的设计目标

一个高质量的操作系统应具有可靠性、高效性、易维护性、可移植性、安全性、可适应性和简明性等特征。

1. 可靠性

可靠性包括了正确性和健壮性。

操作系统是计算机系统中最基本、最重要的软件，随着计算机应用范围的日益扩大，对操作系统的可靠性要求也越来越高。无可靠性，将严重影响使用效果。例如，用于导弹控制的操作系

统必须绝对可靠，否则造成的后果可能不堪设想。

影响操作系统正确性的因素有很多，最主要的因素是并发、共享以及随之带来的不确定性。并发使得系统中各条指令流的执行次序可以任意交叉；共享导致对于系统资源的竞争，使不同的指令执行序列之间产生直接和间接的相互制约；以上两点又引起系统的不确定，这种随机性要求系统能动态地应付随时发生的各种内部和外部事件。因此，需要对于操作系统的结构进行研究。一个设计良好的操作系统不仅应当是正确的，而且其正确性应当是可验证的。

可靠性除了正确性这一基本要求外，还应包括能在预期的环境条件下完成所期望的功能的能力以及在发生硬件故障或某种意外的环境下，操作系统仍能做出适当处理，避免造成严重损失的鲁棒性要求。

2. 高效性

对于支持多道程序设计的操作系统来说，其根本目标是提高系统中各种资源的利用率，即提高系统的运行效率。一个计算机系统在其运行过程中或处于目态，或处于管态。处于目态时为用户服务；处于管态时可能为用户服务（如为进程打开文件或完成打印工作），也可能做系统维护工作（如进程切换、调度页面、检测死锁等）。

假设一个计算机系统，在一段时间 T 之内，目态下运行程序所用的时间为 T_u ，管态下运行程序为用户服务所用的时间为 T_m ，管态下运行程序做系统管理工作所用的时间为 T_{ms} ，则可定义系统运行效率 η 为：

$$\eta = \frac{T_u + T_m}{T_u + T_m + T_{ms}} \times 100\%$$

显然， η 越大，系统运行效率越高。为了提高系统运行效率，应当尽量减少用于系统管理所需要的时间 T_{ms} 。我们亦把 T_{ms} 称为系统开销（时间开销）。

3. 易维护性

易维护性包括易读性、易扩充性、易剪裁性、易修改性等。一个实际的操作系统投入运行后，有时希望增加新的功能，删去不需要的功能，或修改在运行过程中所发现的错误，为了对系统实施增、删、改等维护操作，必须首先了解系统，为此要求操作系统具有良好的可读性。

4. 可移植性

可移植性是指把一个程序从一个计算机系统环境中移到另一个计算机系统环境中并能正常运行的特性。操作系统的开发是一项非常庞大的工程。为了避免重复工作，缩短软件研制周期，现代操作系统设计都将可移植性作为一个重要的目标。而影响可移植性的最大因素就是和机器有关的硬件部分的处理。为了便于将操作系统由一个计算环境迁移到另外一个计算环境中，应当使操作系统程序中与硬件相关的部分相对独立，并且位于操作系统程序的底层，移植时只需修改这一部分。

5. 安全性

操作系统的安全性是计算机软件系统安全性的基础，它为用户数据保护提供了最基本的机制。这一点在网络环境中显得更为重要。

6. 可适应性

可适应性指的是一种特定计算机系统环境中的软件对于另一种计算机系统环境的适应能力。

研制一个大型软件的费用十分昂贵，而使用要求又在不断变化，经常需要对系统做些修改，以适应环境和要求的变化，如果一个系统没有可适应性，它将是一个僵死的系统，是无生命力的。

7. 简明性

无简明性，开发人员就无法了解一个大型程序的设计目的和细节。

具有简明性、可靠性、可适应性的系统称为可维护的系统，我们称之为易管理的。可适应性和可移植性我们合称为灵活性。

2.1.2 操作系统的设计阶段

设计一个操作系统一般可分为三个阶段：功能设计、算法设计和结构设计。

操作系统的三个设计阶段是互相渗透的，因此不能截然分开它们。其总的目的是要求能够设计出一个具有好结构、高功效，又兼备所需功能的系统。

1. 功能设计

功能设计指的是根据系统的设计目标和使用要求，确定所设计的操作系统应具备哪些功能，以及操作系统的类型。

2. 算法设计

算法设计是根据计算机的性能和操作系统的功能，选择和设计满足系统功能的算法和策略，并分析和估算其效能。

3. 结构设计

结构设计则是按照系统的功能和特性要求，选择合适的结构，使用相应方法将系统逐步地分解、抽象和综合，使操作系统结构清晰、简明、可靠、易读、易改，而且使用方便、适应性强。

2.1.3 操作系统的结构问题

1. 程序结构

程序的可靠性和程序结构密切相关。所谓程序结构有两层含义：一是指程序的整体结构，即由程序的成分构造程序的方式，如PASCAL语言程序是分程序结构，EUCLID语言程序是模块结构等；二是指程序的局部结构，即程序的数据结构和控制结构。

我们的目标是设计一个能正确实现功能要求的程序，也就是说，要设计一个可靠的程序，运行的结果能正确地反映设计要求。为此，我们必须构造出一个结构良好的程序。所谓程序的结构良好，指的是程序的结构清晰、易读、易维护、可移植，当然这样的程序也要易验证、易调试和易修改。

结构化程序设计就是为了使程序有一个合理的结构，以便于保证和验证其正确性而规定的一套如何进行程序设计的准则和方法。按照这样一套准则和方法设计出来的程序是结构化程序。

模块化是一种结构化程序设计方法，它指的是把一个程序按功能分解成若干个彼此具有一定独立性，同时也具有一定联系的组成部分，这些组成部分就称作“模块”，每个程序由一个或多个模块组成。对大型程序来说，模块化是一种必然趋势。近年来的研究表明，在分析的基础上设计出一组“基本模块”和一组“构成法则”，然后由这些基本模块出发，通过有关的组装规则，组装成所需的程序。如果这些模块是正确的，而且这些组装规则也是正确的，则得到的程序也是

易于验证其正确性的。

2. 软件结构

软件结构通常是指大型程序系统的结构，与小规模程序结构具有本质的差别，后者主要研究程序的结构良好性、易读性、易验证性等。大型程序是由小规模程序组成，因此要研究由小规模程序组成大型程序。前者是局部结构，后者的结构只有在能保证小规模程序正确性之后，才能使组成的大型程序的正确性有保证。本书讨论的是如何把小规模程序组成大型程序的问题，这些问题在研究小规模程序的结构设计时是不存在的，至少也是不严重的。

一个大型程序系统总是由一些模块组成，模块之间的接口指的是一个模块中的程序访问另一个模块内的程序或数据的方式，也可以说，接口就是指模块间传递和交换信息的方法。设计大型程序系统时，对于接口必须十分重视。

3. 操作系统体系结构

操作系统是一种大型软件。为了研制操作系统，必须分析它的体系结构。也就是要弄清楚如何把这一大型软件划分成若干较小的模块以及这些模块间有着怎样的接口。在操作系统中，有些模块需要使用另一些模块内的数据，而系统的某些功能又需要若干模块协同工作来实现。

例如，有两个模块，一个是命令接收模块A，另一个是命令处理模块B。当命令接收模块A接收到来自操作员的命令后就要把命令交给命令处理模块B去分析执行，于是，模块A和模块B之间就有接口。在这个例子中，模块A要调用模块B的命令分析程序，并要将接收到的命令传送给模块B的命令分析程序。

操作系统还是一个具有并发特性的大型程序，模块间的接口是相当复杂的，信息交换也是十分频繁的，因而对结构的研究就显得更加重要了。

在操作系统的早期设计中，由于计算机体系结构还比较简单，系统规模也比较小，逻辑关系简单，因此人们关心的是系统的功能和效率。随着计算机结构的复杂化，应用范围的不断扩大，使用要求也不断提高，不仅要求有较强的系统功能，而且要求有较强的可适应性和可靠性。后来，又提出了容错的概念。从而使人们日益认识到：体系结构直接影响到整个系统的性能。因此，近年来人们普遍重视操作系统的体系结构和结构设计方法的研究，它已成为软件工程界的一个重要的研究领域。

2.1.4 操作系统的结构设计

在操作系统的发展过程中，产生了多种多样的体系结构，几乎每一个操作系统在结构上都有自己的特点，但从整体上看，到目前为止，大致可划分为四种类型，并且这种划分又和操作系统的发展阶段相一致。但是，这并不意味着后出现的结构已经取代了早期的结构。目前这四种结构的系统都在实际使用中，各有其适用范围。

要说明的是，对于一个实际的现代操作系统来说，它们的体系结构往往比较难划分到某一个单一的类别之下，设计这些系统的工程师们在考虑实际的性能问题时往往会做出各种各样的权衡，将不同的体系结构整合起来，充分吸取它们的优点。

1. 模块组合结构

操作系统是一个有多种功能的系统程序，可以看成是一个整体模块，也可看成是由若干个模块按一定的结构方式组成的。

在1968年软件工程出现以前的早期操作系统（如IBM的操作系统）以及目前的一些小型操作系统（如DOS操作系统）均属此种类型。系统中的模块不是根据程序和数据本身的特性而是根据它们完成的功能来划分，数据基本上作为全程量使用。在系统内部，不同模块的程序之间可以不加控制地互相调用和转移，信息的传递方式也可以根据需要随意约定，因而造成模块间的循环调用，如图2-1。我们把这种操作系统的结构称之为模块组合结构。它的主要优点是：结构紧密、接口简单直接、系统效率较高。它的缺点有以下三点：

- 1) 模块间转接随便，各模块互相牵连，独立性差，系统结构不清晰。
- 2) 数据基本上作为全程量处理，系统内所有模块的任一程序均可对其进行存取和修改，从而造成了各模块间有着很隐蔽的关系。要更换一个模块或修改一个模块都比较困难，因为要弄清各模块间的接口，按当初设计时随意约定的格式来传递信息，这是一件相当复杂的事。
- 3) 由于模块组合结构常以大型表格为中心，因此为保证数据完整性，往往采用全局关中断办法，从而限制了系统的并发性。系统中实际存在的并发也未能抽象出明确的概念，缺乏规格的描述方法。所以，这种结构的可适应性比较差。它只适用于模块比较小、使用环境比较稳定但要求效率比较高的系统。

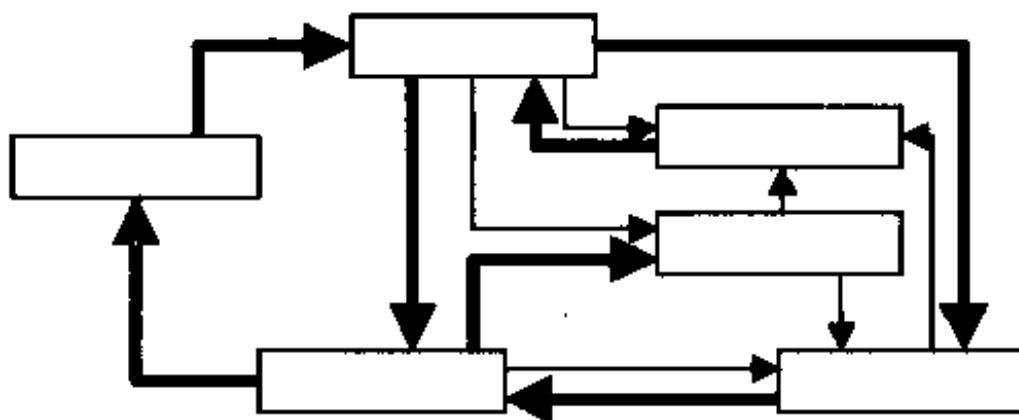


图2-1 模块组合结构

随着系统规模的不断增大，采用这种结构构造的系统的复杂性迅速增长，以致使人们难以驾驭，这就促使人们去寻求新的结构概念和新的结构设计方法。

2. 层次结构

显然，要清除模块接口法的缺点就必须减少各模块之间毫无规则地相互调用、相互依赖的关系，特别是清除循环现象。层次结构设计方法正是从这点出发，它力求使模块间调用的无序性变为有序性。因此所谓层次结构设计方法，就是把操作系统的所有功能模块按功能的调用次序分别排列成若干层，各层之间的模块只能是单向依赖或单向调用（如只允许上层或外层模块调用下层或内层模块）关系。这样，不但操作系统的结构清晰，而且不构成循环，THE系统就是E.W.Dijkstra和他的学生按照层次模型在荷兰的Eindhoven技术学院开发的，如图2-2。

在一个层次结构的操作系统中，如果不仅各层之间是单向调用的，而且每一层中的同层模块之间不存在互相调用的关系，则称这种层次结构关系为全序的层次关系。但是，在实际的大型操

作系统中，要按全序的层次关系来设计几乎是不可能的，往往无法完全避免循环现象，此时我们应使系统中的循环尽量减少。例如，我们可以让各层之间的模块是单向调用的，但允许同层之间的模块互相调用，可以有循环调用现象，这种层次结构关系称为半序的层次结构。

层次结构的优点是：它既具有模块组合结构的优点——把复杂的整体问题分解成若干个比较简单的相对独立的成分，即把整体问题局

部化，使得一个复杂的操作系统分解成许多功能单一的模块；同时它又具有模块组合结构不具有的优点，即各模块之间的组织结构和依赖关系清晰明了。这不但增加了系统的可读性和可适应性，而且还使操作系统的每一步都建立在可靠的基础上。因为层次结构是单向依赖的，所以上一层各模块所提供的功能（以及资源）是建立在下一层的基础上的。或者说上一层功能是下一层的扩充和延续。最内层是硬件基础——裸机，裸机的外层是操作系统的最下面（或内层）的第一层。按照分层虚拟机的观点，每加上一层软件就构成了一个比原来机器功能更强的虚拟机，也就是说进行了一次功能扩充。而操作系统的第一层是在裸机基础上进行的第一次扩充后形成的虚拟机，以后每增加一层软件就是在原机器上的又一次扩充，又成为一个新的虚拟机。因此，只要下层的各模块的设计是正确的，就为上层功能模块的设计提供了可靠基础，从而增加了系统的可靠性。

这种结构的优点还在于增加或替换掉一层可以不影响其他层次，便于修改、扩充。

层次结构的操作系统的各功能模块应放在哪一层，系统一共应有多少层，这是一个很自然会提出的问题。但对这些问题通常并无一成不变的规律可循，必须要依据总体功能设计和结构设计中的功能图和数据流图进行分层，大致的分层原则如下：

1) 为了增加操作系统的可适应性，并且便于将操作系统移植到其他机器上，必须把与机器特点紧密相关的软件（如中断处理、输入输出管理等）放在紧靠硬件的最低层。这样，经过这一层软件扩充后的虚拟机，硬件的特性就被隐藏起来了，方便了操作系统的移植。为了便于修改移植，它把与硬件有关和与硬件无关的模块截然分开，并把与硬件有关的BIOS（管理输入输出设备）放在最内层。所以当硬件环境改变时只需要修改这一层模块就可以了。

2) 对于一个计算机系统来说，往往具有多种操作方式（例如，既可在前台处理分时作业，又可在后台以批处理方式运行作业，也可进行实时控制）。为了便于操作系统从一种操作方式转变到另一种操作方式，通常把多种操作方式共同使用的基本部分放在内层，而把随着这些操作方式而改变的部分放在外层（例如，批作业调度程序和联机作业调度程序、键盘命令解释程序和作业控制语言解释程序等），这样改变操作方式时仅需改变外层，内层部分保持不变。

3) 当前操作系统的设计都是基于进程的概念，进程是操作系统的基本成分。为了给进程的活动提供必要的环境和条件，必须要有一部分软件——系统调用的各功能，来为进程提供服务，通常这些功能模块（各系统调用功能）构成操作系统内核，放在系统的内层。内层中又分为多个层次，通常将各层均要调用的那些功能放在更内层。

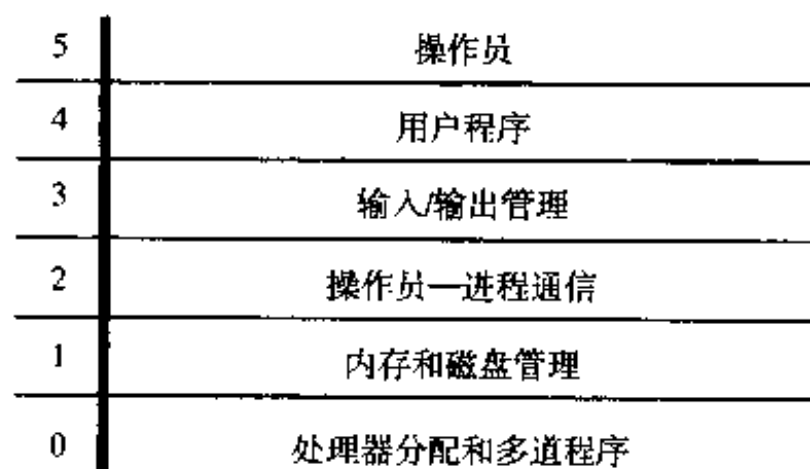


图2-2 THE操作系统的层次结构

3. 虚拟机结构

这是纯粹以虚拟机的观点构造的操作系统体系结构，典型的代表是用在IBM大型机上的系列操作系统OS/370/390/400。

OS/360的最早版本是纯粹的批处理系统。然而有许多希望使用分时系统的360用户，于是IBM公司和另外的一些研究小组决定开发一个分时系统。IBM公司随后提供了一套分时系统TSS/360，它非常庞大，运行缓慢，几乎没有什么人用它。结果在花费了约五千万美元的研制费用后，该系统最终被弃之不用。但是麻省剑桥的一个IBM研究中心开发了另一个完全不同的系统，这个系统被IBM公司最终作为产品，目前仍在IBM公司的大型主机上广泛使用。这个系统最初被命名为CP/CMS，后来改名为VM/370。它基于如下的设计思想，分时系统应该提供这些功能：①多道程序；②一个比裸机更方便扩展界面的计算机。VM/370的任务是将这二者彻底隔离开来。

这个系统的核心被称为虚拟机监控程序，它在裸机上运行并且具备了多道程序功能。该系统向上层提供了若干台虚拟机，如图2-3所示。它不同于其他操作系统的是：这些虚拟机不是那种具有文件等优良特征的扩展计算机。与之相反，它们仅仅是精确复制的裸机硬件。它包含：核心态/用户态；I/O功能；中断；以及其他真实硬件所具有的全部内容。

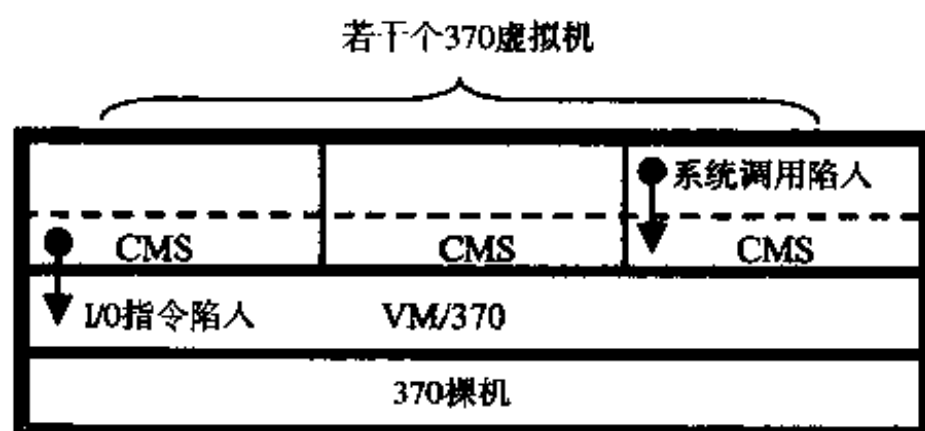


图2-3 带CMS的VM/370体系结构

因为每台虚拟机都与裸机相同，所以每台虚拟机可以运行一台裸机所能够运行的任何类型的操作系统。不同的虚拟机可以运行不同的操作系统，而且实际上往往如此。有一些虚拟机运行OS/360的后续版本，从事着批处理或事务处理；而另一些虚拟机运行单用户、交互式系统，供分时用户们使用，这个系统称作会话监控系统（CMS）。

CMS的程序在执行系统调用时，它的系统调用陷入其虚拟机中的操作系统，而不是调用VM/370，不是在虚拟机上，就像在真正的计算机上一样。然后CMS发出硬件I/O指令，在虚拟磁盘上读或者执行为该系统调用所需的其他操作。这些I/O指令被VM/370捕获，作为对真实硬件模拟的一部分，VM/370随后就执行这些指令。这样，将多道程序的功能和提供扩展机器的功能完全分开后，它们各自都更简单、更灵活和更易于维护。

4. 客户/服务器体系结构

操作系统结构技术的发展是与整个计算机技术的发展相联系的。当前计算机技术发展的突出特点是要求广泛的信息和资源的共享。这一要求促使网络技术的普遍应用和发展。因为网络技术逐渐成熟并实用化，再加上数据库联网已是计算机应用的新趋势，所以为用户提供符合企业

信息处理应用要求的分布式的系统环境是十分必要的。事实上,在一个企业或部门中,数据一般总是在它产生的各个现场上就近被存储、管理、加工、组织和使用,只有少量的数据或加工后的信息才是供全局共享或为局部所使用的。所以,分布式处理才是真正合乎客观实际和新的应用需要的潮流。如果操作系统是采用客户/服务器结构,它将非常适于应用在网络环境下,应用于分布式处理的计算环境中。这种体系结构又被称为微内核的操作系统体系结构,它所具有的一些特征将在下面讨论。

典型的采用客户/服务器结构模式的操作系统有卡内基·梅隆大学研制的Mach和Windows NT的早期版本。它们的共同特点是操作系统由下面两大部分组成:

1) 运行在核心态的内核。它提供所有操作系统基本都具有的那些操作,如线程调度、虚拟存储、消息传递、设备驱动以及内核的原语操作集和中断处理等。这些部分通常采用层次结构并构成了基本操作系统。因为这时的内核只提供一个很小的功能集合,所以通常又称为微内核。

2) 运行在用户态并以客户/服务器方式运行的进程层。这意味着除内核部分外,操作系统所有的其他部分都被分成若干个相对独立的进程,每一个进程实现一组服务,称为服务进程(用户应用程序对应的进程,虽然也以客户/服务器方式活动于该层,但不将其看成操作系统的功能构成成分)。这些服务进程可以提供各种系统功能、文件系统服务以及网络服务等。服务进程的任务是检查是否有客户提出要求服务的请求,并在满足客户进程的请求后将结果返回。而客户可以是一个应用程序,也可以是另一个服务进程。客户进程与服务器进程之间的通信是采用发送消息进行的,这是因为每个进程属于不同的虚拟地址空间,它们之间不能直接通信,必须通过内核进行,而内核则是被映射到每个进程的虚拟地址空间内的,它可以操纵所有进程。客户进程发出消息,内核将消息传给服务进程。服务进程执行相应的操作,其结果又通过内核用发消息方式返回给客户进程,这就是客户/服务器的运行模式。

这种模式的优点在于,它将操作系统分成若干个小的、自包含的分支(服务进程),每个分支运行在独立的用户进程中,相互之间通过规范一致的方式接收发送消息而联系起来。操作系统在内核中建立起最小的机制,而把策略留给在用户空间中的服务进程,这带来了很大的灵活性,直接的好处是:

- 可靠。因为每个分支是独立的和自包含的(分支之间耦合最为松散),所以即使某个服务器失败或产生问题,也不会引起系统其他服务器和系统其他组成部分的损坏或崩溃。
- 灵活。便于操作系统增加新的服务功能,这是因为它们是自包含的,且接口规范。同时修改一个服务器的代码不会影响系统其他部分,可维护性好。
- 适宜于分布式处理的计算环境。由于不同的服务可以运行在不同的处理器或计算机上,从而使操作系统自然地具有分布式处理的能力。

当然这种体系结构也有它的缺陷,主要是对于效率的考虑。因为所有的用户进程只能通过微内核相互通信,所以微内核本身就成为系统的瓶颈,在一个通信很频繁的系统中,微内核往往不能提供很好的效率。例如,高性能的图形用户界面系统中经常有大量的数据在不同的进程中来回拷贝,那么把图形引擎作为一个运行在用户态的服务进程对一个有着高性能图形需求的系统来说将是不明智的选择。

2.2 Windows 2000/XP的操作系统模型

作为一个实际应用中的操作系统，Windows 2000/XP没有单纯地使用某一种体系结构，它的设计融合了分层操作系统和客户/服务器（微内核）操作系统的特点。

Windows 2000/XP像其他许多操作系统一样通过硬件机制实现了核心态（管态，kernel mode）以及用户态（目态，user mode）两个特权级别。当操作系统状态为前者时，CPU处于特权模式，可以执行任何指令，并且可以改变状态。而在后面一个状态下，CPU处于非特权（较低特权级）模式，只能执行非特权指令。一般来说，操作系统中那些至关重要的代码都运行在核心态，而用户程序一般都运行在用户态。当用户程序使用了特权指令，操作系统就能借助于硬件提供的保护机制剥夺用户程序的控制权并做出相应处理。

在Windows 2000/XP中，只有那些对性能影响很大的操作系统组件才在核心态下运行。在核心态下，组件可以和硬件交互，组件之间也可以交互，并且不会引起描述表切换和模式转变。例如，内存管理器、高速缓存管理器、对象及安全管理器、网络协议、文件系统（包括两络服务器和重定向程序）和所有线程和进程管理，都运行在核心态。因为核心态和用户态的区分，所以应用程序不能直接访问操作系统特权代码和数据，所有操作系统组件都受到了保护，以免被错误的应用程序侵扰。这种保护使得Windows 2000/XP可能成为坚固稳定的应用程序服务器，并且从操作系统服务的角度，如虚拟内存管理、文件I/O、网络 and 文件及打印共享来看，Windows 2000/XP作为工作平台仍是稳固的。

Windows 2000/XP的核心态组件使用了面向对象设计原则，例如，它们不能直接访问某个数据结构中由单独组件维护的消息，这些组件只能使用外部的接口传送参数并访问或修改这些数据。但是Windows 2000/XP并不是一个严格的面向对象系统，出于可移植性以及效率因素的考虑，Windows 2000的大部分代码不是用某种面向对象语言写成，它使用了C语言并采用了基于C语言的对象实现。

Windows 2000/XP的最初设计是相当微内核化的，随着不断的改型以及对性能的优化，目前的Windows 2000/XP已经不是经典定义中的微内核系统。出于对效率的考虑，经典的微内核系统在商业上并不具有实践价值，因为它们太低效了。Windows 2000/XP将很多系统服务的代码放在了核心态，包括像文件服务、图形引擎这样的功能组件。应用的事实证明这种权衡使得Windows 2000/XP更加高效而且并不比一个经典的微内核系统更容易崩溃。

Windows 2000/XP的体系结构的框架如图2-4所示，接下来的几小节将对这个图做出详细的说明。

2.2.1 Windows 2000/XP的构成

图2-4中的粗线将Windows 2000/XP分为用户态和核心态两部分。粗线上部的方框代表了用户进程，它们运行在私有地址空间中。用户进程有四种基本类型：① 系统支持进程（system support process），例如登录进程WINLOGON和会话管理器SMSS，它们不是Windows 2000/XP的服务，不由服务控制器启动；② 服务进程（service process），它们是Windows 2000/XP的服务，例如事件日

志服务；③ 环境子系统（environment subsystem），它们向应用程序提供运行环境（操作系统功能调用接口），Windows 2000/XP有三个环境子系统：Win32、POSIX和OS/2 1.2；④ 应用程序（user application），它们是Win32、Windows 3.1、MS-DOS、POSIX 或OS/2 1.2这五种类型之一。

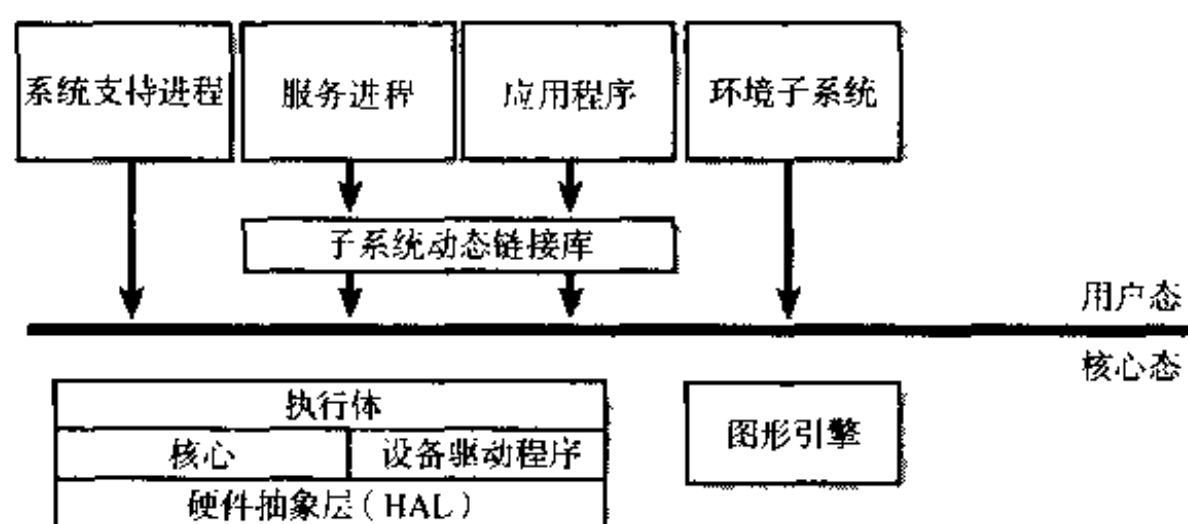


图2-4 Windows 2000/XP体系结构框图

从图2-4中可以看到，服务进程和应用程序是不能直接调用操作系统服务的，它们必须通过子系统动态链接库（subsystem DLLs）和系统交互。子系统动态链接库的作用就是将文档化函数（公开的调用接口）转换为适当的Windows 2000/XP内部系统调用。这种转换可能会向正在为用户程序提供服务的环境子系统发送请求，也可能不会。

粗线以下是Windows 2000/XP的核心态组件，它们都运行在统一的核心地址空间中。核心类组件包括以下内容：① 核心（kernel）包含了最低级的操作系统功能，例如线程调度、中断和异常调度、多处理器同步等，同时它也提供了执行体（Executive）来实现高级结构的一组例程和基本对象；② 执行体包含了基本的操作系统服务，例如内存管理器、进程和线程管理、安全控制、I/O以及进程间的通信；③ 硬件抽象层（Hardware Abstraction Layer, HAL）将内核、设备驱动程序以及执行体同硬件分隔开来，使它们可以适应多种平台；④ 设备驱动程序（Device Drivers）包括文件系统和硬件设备驱动程序等，其中硬件设备驱动程序将用户的I/O函数调用转换为对特定硬件设备的I/O请求；⑤ 图形引擎包含了实现图形用户界面（Graphical User Interface, GUI）的基本函数。

从基本的构成看，Windows 2000/XP和大多数的UNIX系统很相似，它也是一个集成操作系统——它的重要组件和设备驱动程序共享内核受保护的地址空间，任何操作系统组件和设备驱动程序可以很容易地破坏其他组件和驱动程序使用的数据，不过实际中这种事情很少发生。这些重要的系统成分都和应用程序隔离，这种保护使得Windows 2000/XP保持了高效和健壮。

2.2.2 Windows 2000/XP的可移植性

Windows 2000/XP的设计目标之一就是能够在各种硬件体系结构上运行，它用两种方法实现了对硬件结构和平台的可移植性。首先是一个分层的设计，依赖于处理器体系结构或平台的系统底层部分被隔离在单独的模块之中，系统的高层可以被屏蔽在千差万别的硬件平台之外。提供操作系统可移植性的两个关键组件是HAL和内核。依赖于体系结构的功能（如线程描述表切换）在

内核中实现，在相同体系结构中，因计算机而异的功能在HAL中实现。第二个方法是Windows 2000/XP几乎全部使用高级语言写成——执行体、实用程序和设备驱动程序都是用C语言编写的，图形子系统部分和用户界面是用C++编写的。只有那些必须和系统硬件直接通信的操作系统部分（如中断陷阱处理程序），或性能极度敏感（如描述表切换）的部分是用汇编语言编写的。汇编语言代码主要分布在内核及HAL中，极少量分布于执行体的少数区域（例如实现互锁指令的执行体例程）、Win32子系统的核心部分和少数用户态库中，例如在NTDLL.DLL中的进程启动代码。

2.2.3 Windows 2000/XP的对称多处理的支持

Windows 2000/XP支持“对称多处理”（Symmetric MultiProcessing, SMP）。在SMP中不存在主处理器——操作系统和用户线程能被安排在任一处理器上运行；所有的处理器共享一个内存空间。这种模型与“非对称多处理”（ASymmetric MultiProcessing, ASMP）形成对比，后者只能在某个特定处理器上执行操作系统代码，而其他处理器只能运行用户代码。

多处理器系统的一个关键问题是可伸缩性。为了保证系统能在SMP系统上正确运行，操作系统代码必须严格遵守某些规则以确保操作正确。在多处理器系统中，资源竞争及其他性能问题比在单处理器系统中更加复杂，Windows 2000/XP集或了许多关键特性，使之成为一个成功的多处理器操作系统。

Windows 2000/XP能在任何可用的处理器上运行，并且它的完全可重入的代码可以同时多个处理器上运行。当一个较高优先权的线程需要获得处理器时间时，利用系统陷阱调度（trap dispatching）机制，所有操作系统代码都可以被抢先（强制释放一个处理器）。在不同的处理器中，每一个线程基本上都可以同时执行。核心以及设备驱动程序和服务进程内部的精确同步允许更多的组件在多处理器上同时运行，在进程间共享对象的机制及灵活的进程间的通信能力，包括共享内存和优化的消息传送工具。

除了HAL对于单处理器系统和多处理器系统在本质上有不同外，Windows 2000/XP只包含了执行体和内核的核心操作系统映像。NTOSKRNL.EXE这一个文件在单处理器和多处理器版本中是不同的。其他二进制文件则在单处理器和多处理器系统上都能正确运行。

2.3 Windows 2000/XP的体系结构

从本节开始我们将把图2-4的细节逐渐展开（见图2-5），逐一介绍构成Windows 2000/XP的各个组成部分体系结构的细节。

2.3.1 内核

内核执行Windows 2000/XP中最基本的操作，主要提供下列功能：① 线程安排和调度；② 陷阱处理和异常调度；③ 中断处理和调度；④ 多处理器同步；⑤ 供执行体使用的基本内核对象（在某些情况下可以导出到用户态）。

Windows 2000/XP的内核始终运行在核心态，代码短小紧凑，可移植性也很好。一般来说，除了中断服务例程（Interrupt Service Routine, ISR），正在运行的线程是不能抢先内核的。

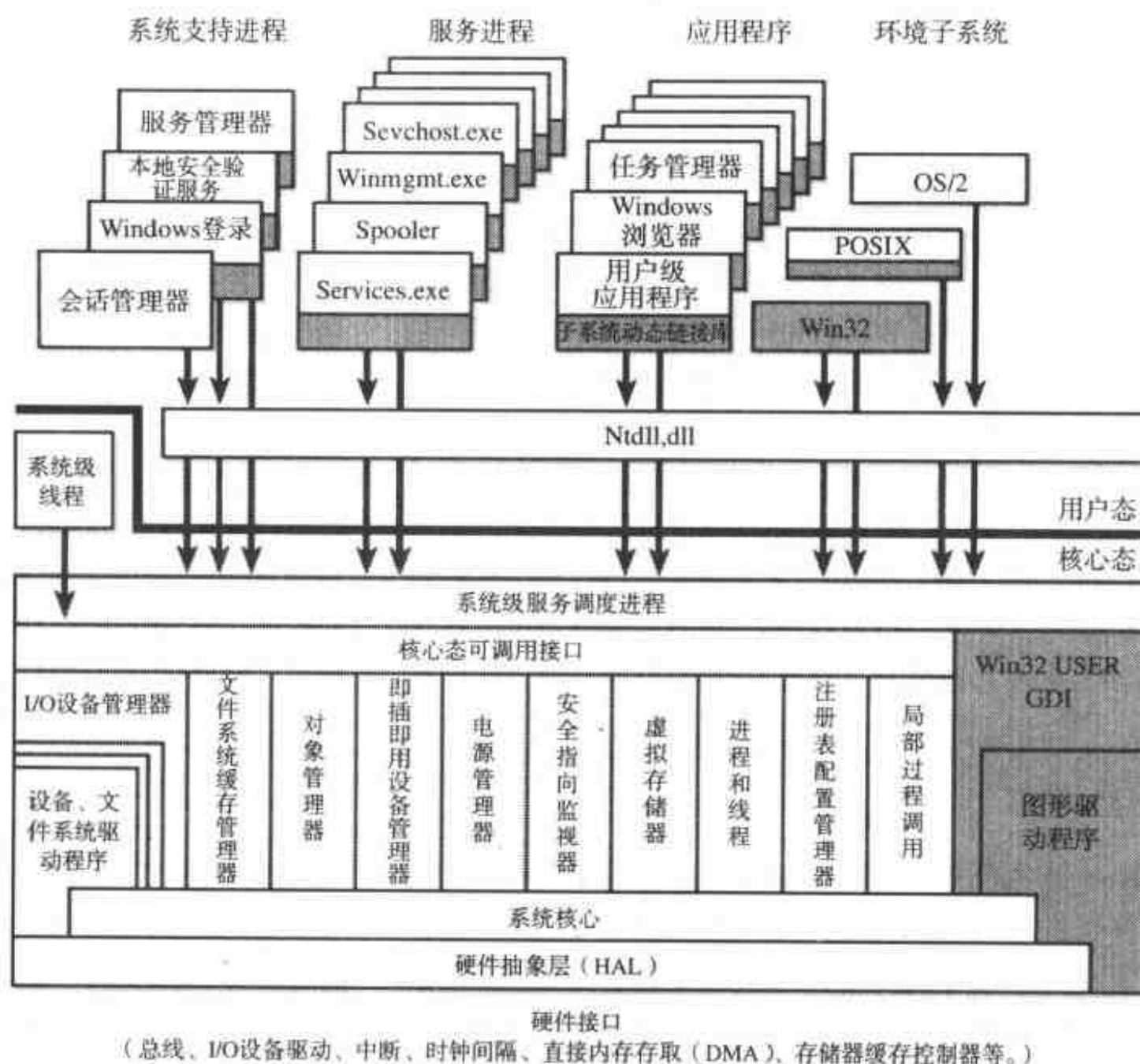


图2-5 Windows 2000/XP的体系结构详图

1. 内核对象

内核提供了一组严格定义的、可预测的、使得操作系统得以工作的基础设施，这为执行体的高级组件提供了必须的低级功能接口。内核除了执行线程调度外，几乎将所有的策略制定留给了执行体。这一点充分体现了Windows 2000/XP将策略与机制分离的设计思想。

在内核以外有很多的系统组件，处理它们的资源分配、安全认证等都要执行体付出不可忽略的策略开销。内核通过一组称作“内核对象”的简单对象帮助控制、处理并支持执行体对象的创建，以降低这种开销。大多数执行体级别的对象都封装了一个或多个内核对象。

一个称作“控制对象”的内核对象集合为控制各种操作系统功能建立了语义。这个对象集合包括内核进程对象、异步过程调用（Asynchronous Procedure Call, APC）对象、延迟过程调用（Deferred Procedure Call, DPC）对象和几个由I/O系统使用的对象（例如中断对象）。

另一个称作“调度程序对象”的内核对象集合负责同步操作并影响线程调度。调度程序对象包括内核线程、互斥体（Mutex）、事件（Event）、内核事件对、信号量（Semaphore）、定时器和可等待定时器。执行体使用内核函数创建内核对象的实例，使用它们来构造更复杂的对象提供

给用户态。

2. 硬件支持

内核的另外一个重要功能就是把执行体和设备驱动程序同硬件体系结构的差异隔离开，包括处理功能之间的差异，例如中断处理、异常情况调度和多处理器同步。对于与硬件有关的函数，内核的设计也是尽可能使公用代码的数量达到最大。内核支持一组在整个体系结构上可移植、语义完全相同的接口，大多数这种接口的实现在整个体系结构上是完全相同的。当然也有一些接口的实现因体系结构而异。Windows 2000/XP可以在任何机器上调用那些独立于体系结构的接口，不管代码是否随体系结构而异，这些接口的语义总是保持不变。一些内核接口实际上是在HAL中实现的，因为同一体系结构内接口的实现可能也因平台系统而异。

内核包含少量支持老版本MS-DOS程序所必需的x86专用代码，这些接口是不可移植的。另一个内核中的体系结构专用代码的例子是提供缓冲区和CPU高速缓存转化支持的接口。因高速缓存执行方式的不同，对于不同的体系结构，这一支持需要的代码也不同。还有就是描述表切换，虽然在更高层次上看来，线程选择和描述表切换使用的是同一种算法，但它们在不同处理器中执行时还是存在结构上的差异。由于描述表是用处理器状态来描述的，因此保存与加载什么取决于体系结构。

2.3.2 硬件抽象层

Windows 2000/XP设计的一个至关重要的方面就是在多种硬件平台上的可移植性，HAL就是使这种可移植性成为可能的关键部分。HAL是一个可加载的核心态模块HAL.dll，它为运行在Windows 2000/XP上的硬件平台提供低级接口。HAL隐藏各种与硬件有关的细节，例如I/O接口、中断控制器以及多处理器通信机制等任何体系结构专用的和依赖于计算机平台的函数。

2.3.3 执行体

Windows 2000/XP的执行体是NTOSKRNL.EXE的上层（内核是其下层）。执行体包括五种类型的函数：

- 1) 从用户态导出并且可以调用的函数。这些函数的接口在NTDLL.DLL中。通过Win32API或一些其他的环境子系统可以对它们进行访问。
- 2) 从用户态导出并且可以调用的函数，但当前通过任何文档化的子系统函数都不能使用。
- 3) 在Windows 2000 DDK中已经导出并且文档化的核心态调用的函数。
- 4) 在核心态组件中调用但没有文档化的函数。例如在执行体内部使用的内部支持例程。
- 5) 组件内部的函数。

执行体包含下列重要的组件，这些组件将在后续的小节中陆续加以介绍：

- 1) 进程和线程管理器创建及中止进程和线程。对进程和线程的基本支持在Windows 2000内核中实现，而执行体给这些低级对象添加附加语义和功能。
- 2) 虚拟内存管理器实现“虚拟内存”。内存管理器也为高速缓存管理器提供基本的支持。
- 3) 安全引用监视器在本地计算机上执行安全策略。它保护了操作系统资源，执行运行时对象

的保护和监视。

4) I/O系统执行独立于设备的输入/输出,并为进一步处理调用适当的设备驱动程序。

5) 高速缓存管理器通过将最近引用的磁盘数据驻留在主内存中来提高文件I/O的性能,并且通过在把更新数据发送到磁盘之前将它们在内存中保持一个短的时间来延缓磁盘的写操作,这样就可以实现快速访问。

另外,执行体还包括四组主要的支持函数,它们由上面列出的执行体组件使用。其中大约有三分之一的支持函数在DDK中已经文档化。这四类支持函数提供下面的功能:

1) 对象管理,创建、管理以及删除Windows 2000/XP的执行体对象和用于代表操作系统资源的抽象数据类型,例如进程、线程和各种同步对象。

2) 本地过程调用(Local Procedure Call, LPC)机制,在同一台计算机上的客户进程和服务进程之间传递信息。LPC是一个灵活的、经过优化的“远程过程调用”(Remote Procedure Call, RPC)版本。

3) 一组广泛的公用运行时函数,例如字符串处理、算术运算、数据类型转换和完全结构处理。

4) 执行体支持例程,例如系统内存分配(页交换区和非页交换区)、互锁内存访问和两种特殊类型的同步对象(资源和快速互斥体)。

2.3.4 设备驱动程序

设备驱动程序是可加载的核心态模块(通常以.SYS为扩展名),它们是I/O系统和相关硬件之间的接口。Windows 2000/XP上的设备驱动程序不直接操作硬件,而是调用HAL功能作为与硬件的接口。

Windows 2000/XP中有如下几种类型的设备驱动程序:① 硬件设备驱动程序操作硬件,它将输出写入物理设备或网络,并从物理设备或网络获得输入;② 文件系统驱动程序接受面向文件的I/O请求,并把它们转化为对特殊设备的I/O请求;③ 过滤器驱动程序截取I/O并在传递I/O到下一层之前执行某些特定处理。

因为安装设备驱动程序是把用户编写的核心态代码添加到系统的唯一方法,所以某些程序通过简单地编写设备驱动程序的方法来访问操作系统内部函数或数据结构,但它们不能从用户态访问。

Windows 2000/XP增加了对即插即用和高级电源选项的支持,它使用Windows驱动程序模型(Windows Driver Model, WDM)作为标准驱动程序模型,同时它也支持Windows NT的驱动程序,不过因为这些驱动不支持即插即用和电源选项,所以使用这些驱动的系统的实际能力将会降低。

从WDM的角度看,有三种驱动程序:

1) 总线驱动程序用于各种总线控制器、适配器、桥或者可以连接子设备的设备,这是必须的驱动程序。

2) 功能驱动程序用于驱动那些主要的设备,提供设备的操作接口。一般来说,这也是必须的,除非采用一种原始的方法来使用这个设备(功能都被总线驱动和总线过滤器实现了,例如SCSI PassThru)。

3) 过滤器驱动程序用于为一个设备或者一个已经存在的驱动程序增加功能，或者改变来自其他驱动程序的I/O请求和响应行为。过滤器驱动程序是可选的，并且可以有任意的数目，它存在于功能驱动程序的上层或者下层、总线驱动程序的上层。

在WDM的驱动程序环境中，没有一个单独的设备驱动控制着某个设备。总线设备驱动程序负责向即插即用管理器报告它上面有的设备，而功能驱动程序则负责操纵这些设备。

2.3.5 环境子系统和子系统动态链接库

Windows 2000/XP有三种环境子系统：POSIX、OS/2和Win32（OS/2只能用于x86系统）。在这三个子系统中，Win32子系统比较特殊，如果没有它，Windows 2000/XP就不能运行。而另外两个子系统只是在需要时才被启动，而Win32子系统必须始终处于运行状态。

环境子系统的作用是将基本的执行体系统服务的某些子集提供给应用程序。每个子集都可以提供访问Windows 2000/XP中本地服务的不同子集，函数调用不能在子系统之间混用。用户应用程序不能直接调用Windows 2000/XP系统服务，这种调用必须通过一个或多个子系统动态链接库作为中介才可以完成。例如，Win32子系统动态链接库（如KERNEL32.DLL、USER32.DLL和GDI32.DLL）实现Win32API函数，POSIX子系统动态链接库则实现POSIX 1003.1API。

每一个可执行的映像（.EXE）都受限于唯一的子系统，进程创建时，程序映像头中的子系统类型代码会告诉Windows新进程所属的子系统。类型代码可以使用Windows 2000资源管理器中内置的快速查看器、Link/DUMP命令或者在Windows 2000资源工具包中的Exetype工具来查看。

当一个应用程序调用子系统动态链接库中的函数时，会出现下面三种情况之一：

1) 函数完全在子系统动态链接库的用户态部分中实现，这时并没有消息发送到环境子系统进程，也没有调用执行体服务。函数在用户态中执行，结果返回到调用者。

2) 函数需要一个或多个对执行体的调用。

3) 函数要求某些工作在环境子系统进程中进行。在这种情况下，将产生一个客户/服务器请求到环境子系统，其中的一个消息将被发送到子系统去执行某些操作，这可能会使用执行体的“本地过程调用”（LPC）机制。然后，子系统动态链接库在消息返回给调用者之前会一直等待应答。

此外，某些函数可能是上述第二与第三项的结合，如Win32 Create Process和Create Thread函数。

Windows 2000/XP可以支持多重独立环境子系统，但从实用角度来看，每个子系统执行所有的代码并处理窗口和显示I/O将有大量系统函数的重复，这很可能对系统大小和性能产生负面影响。因而，Windows 2000/XP中Win32是主子系统，基本函数都放在该子系统中，并且让其他子系统调用Win32子系统来执行显示I/O。这样，POSIX与OS/2将调用Win32子系统中的服务来执行显示I/O。目前的POSIX子系统只能执行一个非常有限的函数集（仅POSIX 1003.1），这对于移植UNIX应用程序来说并不是一种有用的环境。在Windows XP中，这两个部分实际上已经被去除了，其中POSIX子系统将在改型优化以后出现在微软公司更加新的系统中。

1. Win32子系统

Win32子系统由下列重要组件构成：

- Win32环境子系统进程CSRSS，包括对下列功能的支持：控制台（文本）窗口、创建及删除进程与线程、支持16位DOS虚拟机（VDM）进程的部分。
- 其他混杂的函数，如GetTempFile、DefineDosDevice、ExitWindowsFx和几种自然语言支持函数。
- 核心态设备驱动程序（WIN32K.SYS），包括下列功能：窗口管理器控制窗口显示；管理屏幕输出；收集来自键盘、鼠标和其他设备的输入信息；以及将用户信息传送给应用程序。
- 图形设备接口（Graphics Device Interface, GDI）是一个用于图形输出设备的函数库，它包括线条、文本、绘图和图形操作函数。
- 子系统动态链接库（例如USER32.DLL、ADVAPI32.DLL、GDI32.DLL和KERNEL32.DLL），它调用NTOSKRNL.EXE和WIN32.SYS将文档化的Win32 API函数转化为适当的非文档化的核心系统服务。
- 图形设备驱动程序，它包括依赖于硬件的图形显示驱动程序、打印机驱动程序和视频小端口驱动程序。

应用程序调用标准的USER函数在显示器上创建窗口和按钮。窗口管理器传递这些请求到GDI，GDI再将这些请求传送给图形设备驱动程序，在这里将按照显示设备的要求将其规格化。显示驱动程序与视频小型端口驱动程序相配合来完成对视频显示的支持。每个视频小端口驱动程序都对与之相关的显示驱动程序提供硬件级支持。

GDI提供了一组标准的函数，它使得应用程序可以同图形设备（包括显示器和打印机）通信而不必知道关于这些设备的任何事情。GDI的各种函数在应用程序与图形设备（例如显示驱动程序及打印机驱动程序）之间起协调作用。GDI解释应用程序对图形输出的要求，并把它们发送到图形显示驱动程序。GDI也能够为应用程序提供使用不同图形输出设备的标准接口。这个接口可以让应用程序代码独立于硬件设备和硬件设备驱动程序。GDI为使其信息适合设备的功能，常常把要求划分为易于处理的各个部分。

对于在客户端的每一个线程，这里都有专用的成对的服务器线程在Win32子系统进程中等待客户进程的请求。一个称作“快速LPC”的在进程间通信的特殊机制在这些线程间发送信息。同正常的线程描述表切换不同，通过快速LPC在一对线程之间进行的转换不会在内核中产生再调度事件，这样客户线程在内核抢先线程调度程序中获得它的时间片之前，允许服务线程在客户线程的剩余时间片中运行。此外，共享内存缓冲器被用作允许快速传递一些大的数据结构（例如位图），同时为了最小化在客户和Win32服务之间线程/进程转换的要求，客户可以直接（但是以只读方式）访问关键的服务器数据结构。GDI操作也是批处理化的（现在仍然是）。“批处理”意味着一系列由Win32应用程序调用的图形不会被“推”到服务器上，也不会被画到输出设备上，直到一个GDI批处理队列被装满。您可以使用Win32 GdiSetBatchLimit函数设置队列的大小，也可以在任何时候使用GdiFlush刷新队列。相反，对于GDI的只读属性和数据结构，一旦从Win32子系统进程得到它们，它们就会在客户端被高速缓存以用于快速后继访问。

尽管采用了这些优化，整个系统性能仍然不能满足图形密集型应用程序的要求。最明显的解决方案就是通过将窗口和图形系统移入到核心态来消除对附加线程和因此带来的对描述表切换的要求。同样，一旦应用程序调用进入到窗口管理器和GDI中，这些子系统就可以直接访问其他执行体组件，而不会有用户态或核心态转换的费用。在通过GDI调用视频驱动程序（一个进程，它涉及与视频硬件在高频、高带宽下的交互作用）的情况下，这种直接访问尤其重要。

Win32子系统的用户态进程部分还包括所有对控制台或文本窗口的描绘和更新。但是除了控制台窗口支持以外，只有少数Win32函数会给Win32子系统进程发送消息：进程和线程的创建和中止、映射网络驱动器以及创建临时文件。一般来说，正在运行的Win32应用程序不会引起太多到Win32子系统进程的描述表切换。

2. POSIX子系统

POSIX代表了UNIX类型的操作系统接口的国际标准集，它鼓励制造商实现兼容的UNIX风格接口，以使编程者能够很容易地将他们的应用程序从一个系统移到另一个系统。Windows 2000/XP只实现了POSIX.1标准（ISO/IEC 9945-1 1990或IEEE POSIX 1003.1-1990）。所需的POSIX一致性文档位于Platform SDK中的\HELP目录中。

Windows 2000/XP被设计成确保提出所需的基本操作系统支持以便考虑POSIX.1子系统的实现，然而，因为POSIX.1只定义了一组有限的服务，所以单独的POSIX子系统并不是一个完整的编程环境。并且因为在Windows 2000/XP上应用程序不能在子系统之间混合调用，所以POSIX应用程序被严格限制在POSIX.1中定义的一组服务。这种限制意味着在Windows 2000/XP上可执行的POSIX不能创建线程或窗口，也不能使用远程过程调用（RPC）或套接字，然而您可以在Win32应用程序中做所有这些事情。

不过，这种情况在下一代的Windows（Interix）操作系统中将有所改变。我们可以看一下Interix的体系结构图（图2-6、图2-7）。

从图2-6可以知道，在Interix的计划中，POSIX/UNIX子系统得到了相当大的改善，支持了主流的UNIX标准和UNIX应用（例如X以及标准UNIX Shell）。它几乎成为和Win32子系统平行的部分。

3. OS/2子系统

OS/2子系统在实用性方面受到很大的限制，它仅支持X86系统以及基于16位字符的OS/2 1.2或视频I/O应用程序。

4. NTDLL.DLL

NTDLL.DLL是一个特殊的系统支持库，主要用于子系统动态链接库。NTDLL.DLL包含两种类型的函数：执行体提供的系统服务调度占位程序；子系统、子系统动态链接库以及其他本机映像使用的内部支持函数。其中第一组函数提供了可以从用户态调用的作为Windows 2000/XP执行体系统服务的接口。这些函数的大部分功能都可以通过Win32 API访问。对于这些函数中的每个函数，NTDLL都包含一个有相同名称的入口点。在函数内的代码含有体系结构专用的指令，它能够产生一个进入核心态的转换以调用系统服务调度程序。在进行一些验证后，系统服务调度程序将调用包含在NTOSKRNL.EXE内的实代码的实际的核心态系统服务。NTDLL也包含许多支持

函数，例如映像加载程序、堆管理器和Win32子系统进程通信函数以及通用运行时库例程。此外，它还包含用户态异步过程调用（APC）调度器和异常调度器。

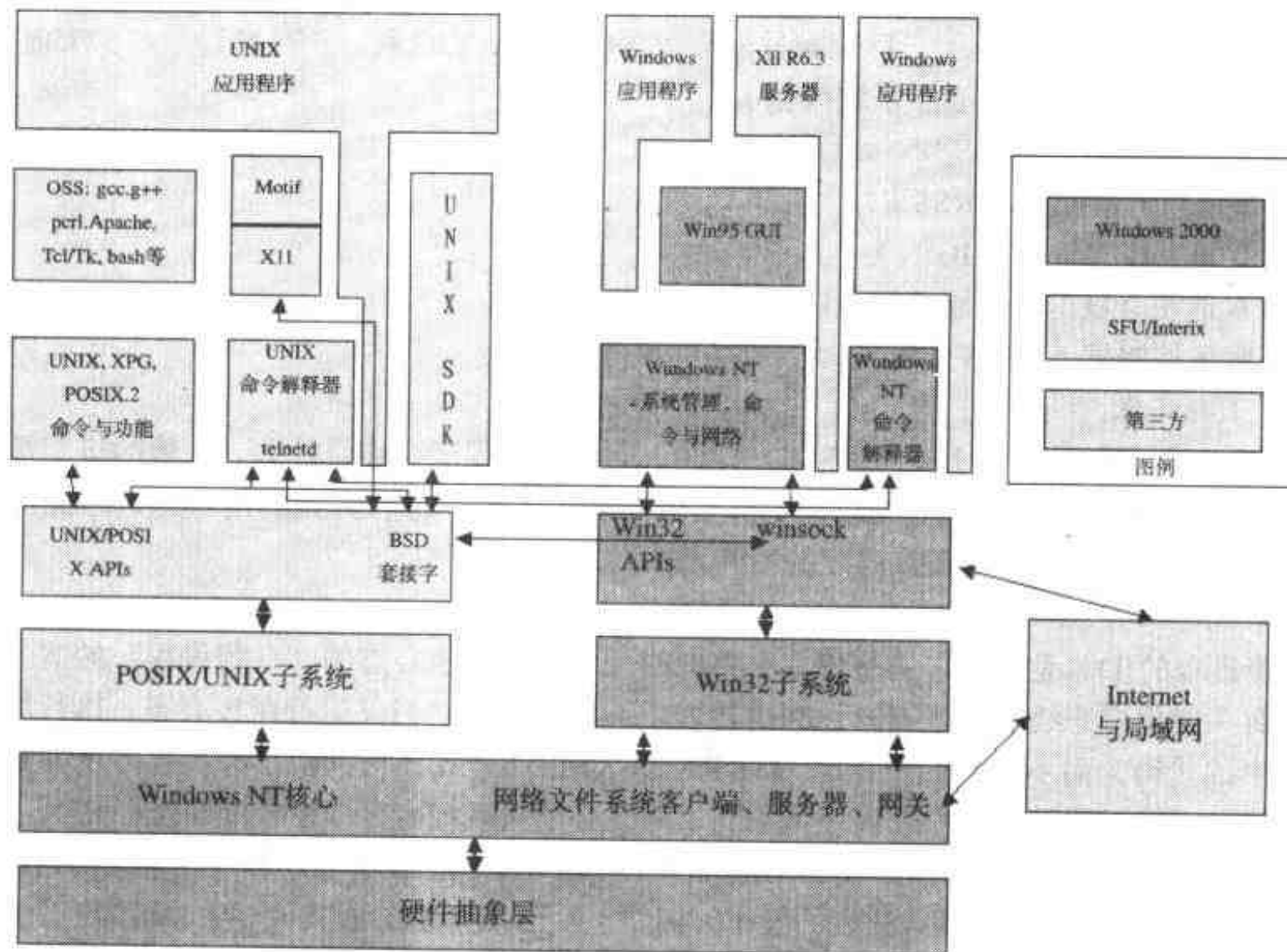


图2-6 Interix的体系结构

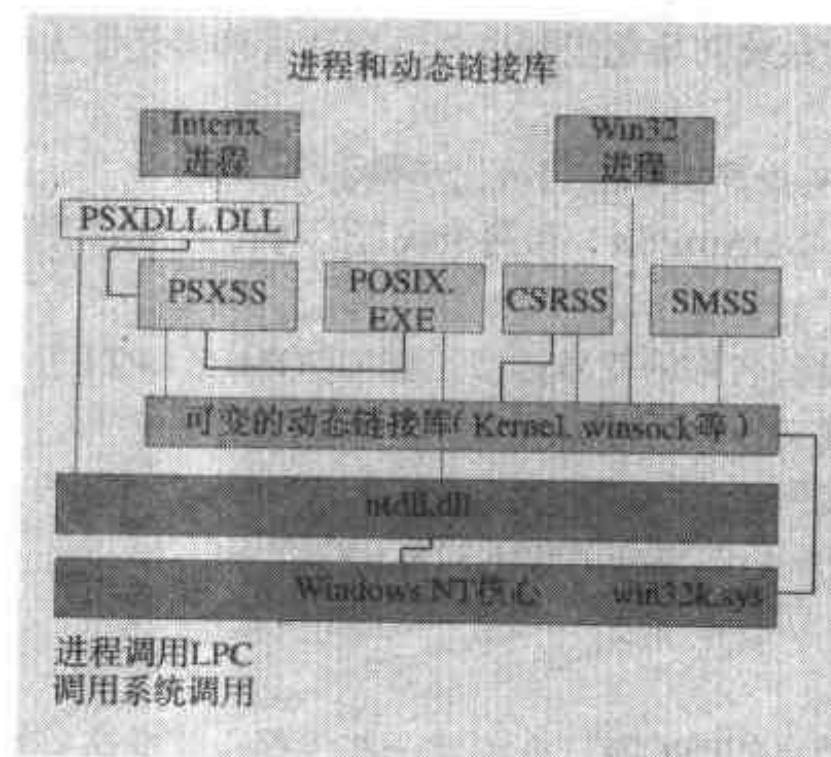


图2-7 Interix的调用机制

2.3.6 系统支持进程

下面的系统支持进程在所有的Windows 2000/XP系统上都有。

- Idle进程（对于每个CPU，Idle进程都包含一个相应的线程，用来统计空闲CPU时间）。
- 系统进程（包含核心态系统线程）。
- 会话管理器（SMSS）。
- Win32子系统（CSRSS）。
- 登录进程（WINLOGIN）。
- 本地安全身份验证服务器（LSASS）。
- 服务控制器（SERVICES）及其相关的服务进程。

1. Idle进程

在Windows 2000/XP中Idle进程的ID总是0，而不管进程的名称是什么。一般的进程都有它们的映像名标识，Idle（以及进程ID2，名称是System）不是运行在真正的用户态，因此由不同的进程观察程序显示的名称是随该程序的不同而不同的值。

2. 系统进程和系统线程

系统进程的ID总是2，它是一种特殊类型的、只运行在核心态的“系统线程”的宿主。系统线程具有一般用户态线程的所有属性和描述表，不同点在于它们仅运行在核心态，执行加载于系统空间中的代码，而不管它们是在NTOSKRNL.EXE中还是在任何其他已经加载的设备驱动程序中。另外，系统线程没有用户进程地址空间，因此必须从系统内存堆中分配动态存储区。

系统线程只能从核心态调用。Windows 2000/XP以及不同的设备驱动程序在系统初始化时创建系统线程以执行那些需要线程描述表的操作，例如，发布和等待I/O或其他对象，轮询一个设备等。

3. 会话管理器

会话管理器是第一个在系统中创建的用户进程。由核心系统线程运行例程ExInitializeSystem创建。除了执行一些关键的系统初始化步骤以外，会话管理器还作为应用程序和调试器之间的开头和监视器。

下面列出了由SMSS的主线程执行的初始化步骤：

- 1) 创建LPC端口对象（\SmApiPort）和两个线程，等待客户的请求。
- 2) 创建系统环境变量。
- 3) 定义用于MS-DOS设备名称的符号链接（例如COM1或LPT1）。
- 4) 创建附加的页面调度文件。
- 5) 打开已知的动态链接库。
- 6) 加载Win32子系统的核心态部分（WIN32K.EXE）。
- 7) 启动子系统进程。
- 8) 启动登录进程。
- 9) 创建用于调试事件消息的LPC端口并创建一些线程来监视这些端口。

在执行这些初始化步骤之后，SMSS中的主线程将永远等待CSRSS和WINLOGON的进程句

柄，如果这些进程意外终止，SMSS将使系统崩溃。

当然，SMSS中的其他线程会负责把消息发送给上面提到的LPC端口，例如请求加载子系统、启动新的子系统和调试事件。

4. 登录进程

Windows 2000的登录进程WINLOGON处理用户登录和注销的内部活动。当输入“安全注意序列”(SAS, Security Attention Sequence, 常为Ctrl + Alt + Del)的组合键时，用户登录请求就通知WINLOGON，使用SAS的原因是保护用户不受那些能模拟登录进程的密码捕获程序的干扰。一旦用户名和密码被捕获，它们将被发送到本地安全身份验证服务器进程以确认其合法性。如果确认相符，将创建一个叫做USERINIT.EXE的进程，这个线程在注册表中查找并创建系统定义的Shell，随后USERINIT就退出了。

登录进程的标识和身份验证是在名为GINA的可替换动态链接库中实现的。作为标准Windows 2000/XP的GINA DLL, MSGINA.DLL来实现其他的标识和身份验证机制以代替标准的Windows 2000/XP用户名/密码方法。另外，WINLOGON可以加载附加的需要执行二级身份验证的网络提供者动态链接库。这种能力允许多个网络提供者收集所有的在一次正常登录时的标识和身份验证信息。

WINLOGON不仅在用户登录和注销时是活动的，无论何时从键盘截取SAS，它也是活动的。

5. 本地安全身份验证服务器

本地安全身份验证服务器进程接收来自WINLOGON的身份验证请求，并调用适当的身份验证包来执行实际的验证，例如检查一个密码是否与存储在SAM文件中的密码匹配。

在身份验证成功时，LSASS将生成一个包含用户安全配置文件的访问令牌对象。WINLOGON随后使用这个访问令牌去创建初始外壳进程。这些进程将从外壳启动，然后默认地继承这个访问令牌。

6. 服务控制器

在Windows 2000中，“服务”既可以指服务进程也可以指设备驱动程序，这里特指用户态进程服务。服务就像UNIX的“守护进程”或VMS的“派遣进程”一样，可以配置成在系统引导时自动启动而不需要交互式登录，当然，服务也可以手工启动。

服务程序是真正合法的Win32映像，这些映像调用特殊的Win32函数以与服务控制器相互使用，例如注册、启动、响应状态请求、暂停或关闭服务。一些Windows 2000组件是作为服务来实现的，例如假脱机、事件日志、用于RPC的支持和其他各种各样的网络组件。

服务由服务控制器启动和停止。服务控制器是一个运行映像为SERVICES.EXE的特殊系统进程，它负责启动、停止和与服务控制器交互。

2.4 Windows 2000/XP的系统机制

在上一节中，我们已经了解了Windows 2000/XP基本的组成，从这一节开始我们将讨论有关这个体系结构如何运作的问题。Windows 2000/XP提供了核心态组件工作的基本机制：

- 陷阱调度 (Trap Dispatching)，包括中断调度 (Interruption Dispatching)、延迟过程调用

(Deferred Procedure Call, DPC)、异步过程调用 (Asynchronous Procedure Call, APC)、异常调度 (Exception Dispatching) 和系统服务调度 (System Service Dispatching)。

- 执行体对象管理器 (Executive Object Manager)。
- 同步 (Synchronization), 包括自旋锁 (Spinlock)、内核调度程序对象 (Kernel Dispatcher Objects)。
- 其他方面的机制, 如Window NT 全局标志。
- 本地过程调用 (LPC, Local Procedure Call)。

2.4.1 陷阱调度

陷阱处理器提供给操作系统用来处理意外事件的硬件机制。当异常或中断发生时, 硬件或软件可以检测到它们, 并捕获正在执行的线程, 处理器会从用户态切换到核心态, 并将控制转交给操作系统中相对固定的地址。在Windows 2000/XP中, 处理器将控制转交给内核的陷阱处理程序, 该模块检测异常和中断的类型, 并将控制交给处理相应情况的代码。图2-8给出了Windows 2000/XP陷阱调度的框架。

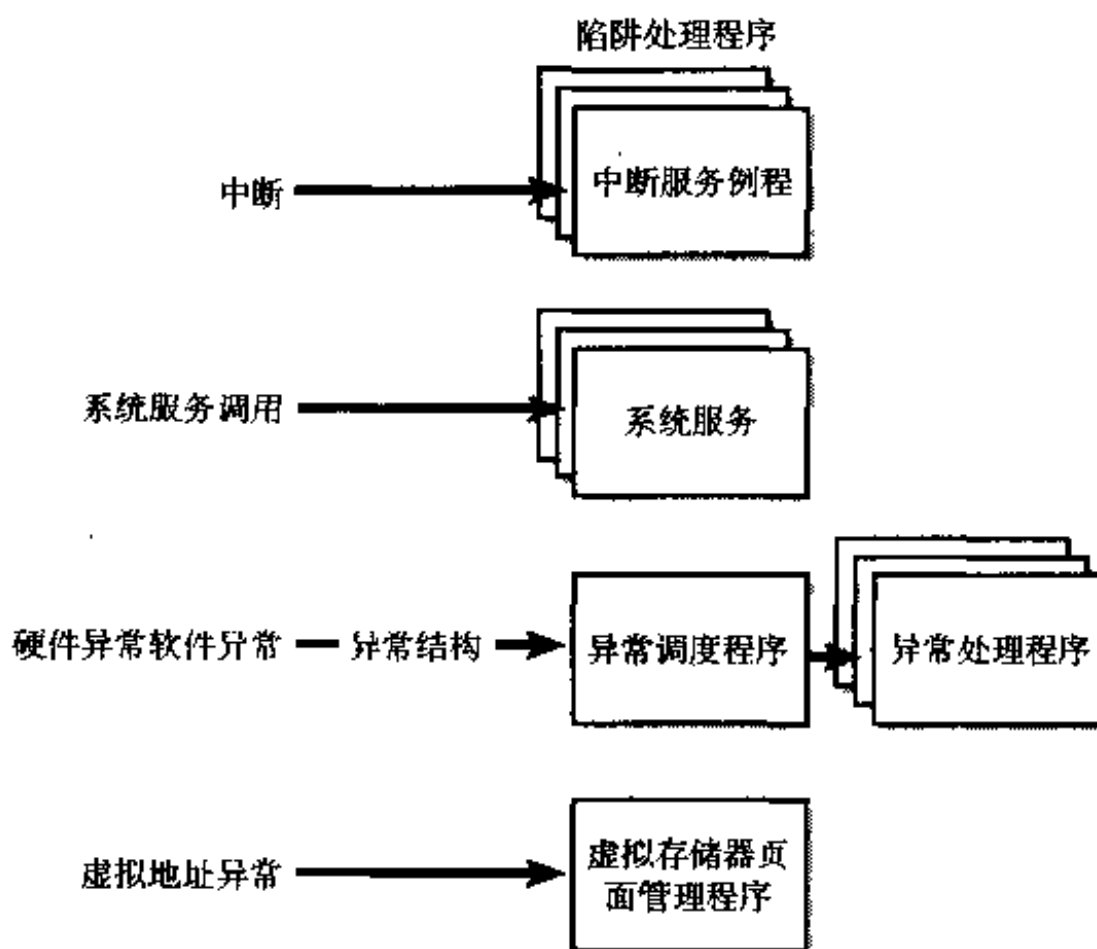


图2-8 Windows 2000/XP陷阱调度框架

一般来说, 中断是异步事件, 可能随时发生, 与处理器正在执行的内容无关。中断主要由I/O设备、处理器时钟或定时器产生, 可以被启用或禁用; 异常是同步事件, 它是某一特定指令执行的结果。在相同条件下, 异常可以重现。例如, 内存访问错误、调试指令以及被零除。系统服务调用也视作异常。软件和硬件都可以产生异常和中断。例如, 总线错误异常是由硬件问题造成的, 而被零除异常则是由软件错误引起的。同样, I/O设备可以产生中断, 内核自身也可以发出软件中断。

当陷阱处理程序被调用时，将在记录机器状态时暂时禁用中断。它会创建一个陷阱帧（Trap Frame）来保存被中断线程运行现场，这用来在合适的时候恢复线程的执行。陷阱帧通常是完整的线程描述表的子集。陷阱处理程序本身可以处理一些事件，但大多数情况下，陷阱处理程序判定发生的情况，并将控制转交给其他的内核或执行体模块。例如，如果情况是设备中断产生的，内核把控制转交给设备驱动程序提供给该中断设备的中断服务例程（ISR）。如果情况是由调用系统服务产生的，陷阱处理程序会将控制转交给执行体中的系统服务代码。

1. 中断调度

最典型的硬件中断是由I/O设备产生的，当这些设备需要服务时，必须通知处理器。中断驱动的设备允许操作系统通过将指令执行与I/O操作重叠进行来获得处理器的最大利用率。处理器启动发往设备的I/O传送或来自设备的I/O传送，然后在设备完成传送时执行其他线程。当设备执行完后，它中断处理器以获得服务。定点设备、打印机、键盘、磁盘驱动器以及网卡通常都是中断驱动的。

软件也可以产生中断，例如，内核可以发布启动线程调度的软件中断。内核也可以禁用中断以使处理器不被中断，但这种情况很少出现，只在处理中断或调度异常的关键时刻才这样做。

中断由中断调度程序的子模块响应。它确定中断源并将控制转交给处理中断的外部例程（ISR），或转交给响应中断的内核例程。设备驱动程序给服务设备中断提供ISR，内核则提供其他类型中断的中断处理例程。

(1) 中断类型和优先级

不同的处理器中断机制也不一样，Windows 2000/XP的中断调度程序将硬件中断级映射到由操作系统识别的中断请求级别（Interrupt ReQuest Level, IRQ Level）的标准集上（如图2-9）。

IRQ Level与线程的调度优先级具有完全不同的含义。调度优先级是线程的属性，而IRQ Level是中断源的属性，每个处理器都具有一个IRQ Level设置，其值随着操作系统代码的执行而改变。

内核定义了一组可移植的IRQ Level，如果处理器具有与中断相关的特性，则可以增加IRQ Level。IRQ Level按优先级排列中断，并进行中断服务，较高优先级的中断服务可以抢占较低优先级的中断服务。

IRQ Level从高往低到设备都是为硬件中断保留的。Dispatch/DPC和APC级中断是内核和设备驱动器产生的软件中断（DPC和APC在本章稍后将详细介绍）。低优先级（也称作被动级）实际上并不是真正的中断级，在该级上执行普通的线程，并允许发生所有的中断。

IRQ Level设置决定了每个处理器可以接收的中断。当核心态线程运行时，它可以提高或降低处理器的IRQ Level来屏蔽一些事件。如果中断源的IRQ Level高于当前中断设置，则它的中断可以中断该处

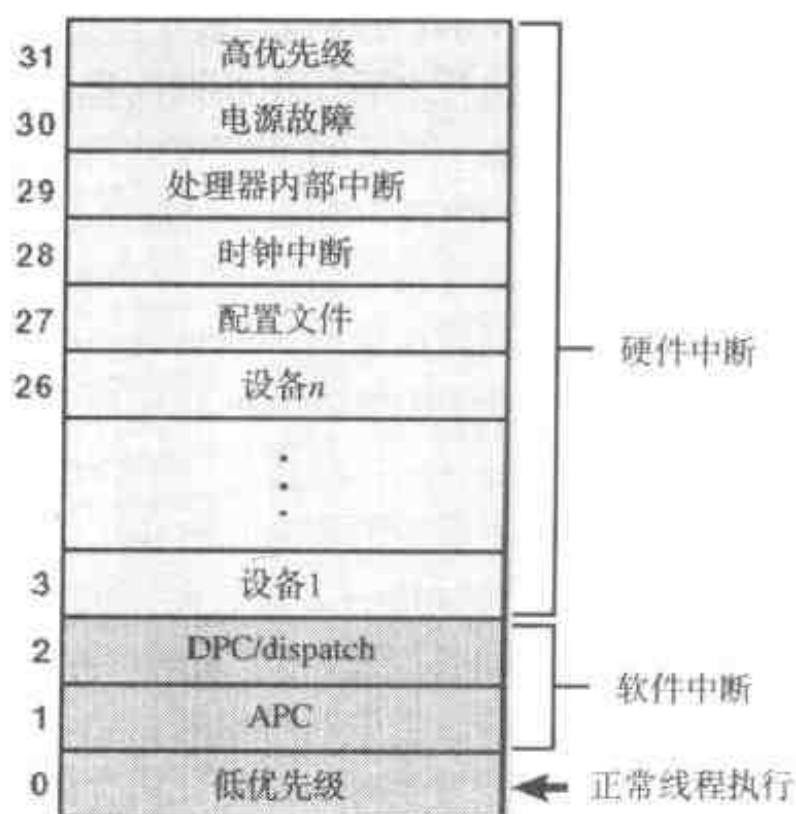


图2-9 中断请求级别

理器；如果中断源的IRQL等于或低于当前中断设置，则它的中断将被封锁或“屏蔽”，直到一个正在执行的线程降低了IRQL。当产生中断时，陷阱处理程序提高处理器的IRQL直到与中断源所指定的IRQL相同，这可以保证服务于该中断的处理器不会被同级或较低级的中断抢先。被屏蔽的中断将被另一个处理器处理或阻挡，直到IRQL降低。因为改变处理器的IRQL对操作系统具有如此重要的影响，所以它只能在核心态下改变。

每个中断级都有特定的用途。例如，内核为了请求另一个处理器执行某个操作而发出处理器间中断；系统时钟以一定的间隔产生中断，内核通过更新时钟和测量线程的执行时间做出响应；HAL提供了许多中断驱动设备使用的中断级，确切数目依据处理器与系统配置而定；内核使用软件中断可启动线程调度并异步中断线程的执行。

(2) 硬件中断处理

当中断产生时，陷阱处理程序将保存计算机的状态，然后禁用中断并调用中断调度程序。中断调度程序立刻提高处理器的IRQL到中断源的级别，以便在中断服务进行时屏蔽等于或低于中断源级别的中断。然后，重新启用中断，以使高优先级的中断仍然能够得到服务。

Windows 2000 使用中断分配表（Interrupt Dispatch Table, IDT）来查找处理特定中断的例程。中断源的IRQL作为表的索引，表的入口指向中断处理例程。在x86系统中，外部中断源触发中断控制器去中断处理器，一旦处理器被中断，它将向中断控制器询问中断向量。处理器利用此向量索引查询硬件IDT并将控制转交给适当的中断调度例程。

在服务例程执行之后，中断调度程序将降低处理器的IRQL到该中断发生前的级别，然后加载保存的机器状态，中断线程将从它停止的地方继续执行。在内核降低了IRQL后，被封锁的低优先级中断就可能出现。在这种情况下，内核将重复以上过程处理新的中断。

每个处理器都有独立的IDT，不同的处理器可以运行不同的ISR。例如，在多处理器系统中，每个处理器都收到时钟中断，但只有一个处理器在响应该中断时更新系统时钟。然而，所有的处理器都使用该中断来测量线程的时间片并在线程时间片结束后做一次调度。同样地，某些系统配置可能要求特殊的处理器处理某一设备中断。

大多数处理中断的例程都在内核中，为了管理它们，内核提供了称为中断对象的内核控制对象。中断对象是可移植的，并且允许设备驱动程序注册其设备的ISR。中断对象包含内核所需的将设备ISR与中断的特定级相联系的所有信息，包括ISR的地址、设备中断的IRQL以及与ISR相联系的内核中的入口。当中断对象被初始化后，称为调度代码的一些汇编语言代码指令就会被存储在对象中。当中断发生时，这些代码会调用真正的中断调度程序，并传递一个指向中断对象的指针。中断对象包含了第二个调度程序例程所需要的信息，以便定位和正确地调用设备驱动程序提供的ISR。需要两步过程的原因是自硬件完成初始调度后，没有一种方法可以在初始调度上传递一个指向中断对象的指针。

把ISR与某个中断级相关联叫做“连接一个中断对象”，而从IDT入口分离ISR叫做“断开一个中断对象”。这些操作允许设备驱动程序在被加载到系统时“打开”ISR，在卸载驱动程序时“关闭”ISR，它们可以通过调用内核函数来完成。

使用中断对象来注册ISR，可防止设备驱动程序直接随意中断硬件，并使设备驱动程序无需

了解IDT的任何细节。内核的这个特性有助于创建可移植的设备驱动程序，这是因为它消除了设备驱动程序中体现处理器差异的需要。

中断对象使内核更容易调用多个任意中断级的ISR。如果多个设备驱动程序创建多个中断对象并将它们连结到同一个IDT入口，那么当中断在指定的中断级上发生时，中断调度程序会调用每一个例程。这样就使得内核很容易地支持“菊花链”配置，在这种构造中几个设备在相同的中断行上中断。

中断处理的另一个考虑是有关实时性的。时限要求是所有实时环境的共同特征，核反应堆控制系统等硬实时系统必须满足所设定的时限要求，以避免设备或生命损失等巨大的灾难性后果。轿车燃料控制系统等软实时系统的时限要求是可以错过的，但实时性仍然是系统追求的重要特征。实时系统中的计算机一般都有传感输入设备和控制输出设备。实时计算机系统的设计者必须知道，从输入设备产生中断请求到设备驱动程序控制输出设备作出响应之间的最长延时。这种延时分析必须考虑到操作系统、应用程序和驱动程序所占用的时间。

由于Windows 2000/XP并不以任何可控制的方式区分设备中断请求的优先顺序，用户级应用程序只能在处理器的被动中断优先级执行，因此它并不总是一个合适的实时操作系统。这不是由操作系统决定，而是由系统中的设备和设备驱动程序决定最坏情况下的延迟时间。当设计者希望使用平台无关的硬件时，这就成了一个很大的问题，因为无法决定每一个硬件的ISR或者DPC需要多长时间。即使经过测试，设计者也不能保证在实际系统中不会由于某种特殊情况而超过一个重要的时间限制。所有系统设备的DPC和ISR延时总和也大大超过了时间敏感系统的时限要求。

尽管打印机、车载计算机等许多嵌入式操作系统都有实时的要求，但Windows NT的嵌入式版本并不具有实时特性。它只是一个简化的Windows NT版本，该版本是微软公司利用VenturCom的技术设计完成的，适合于运行在资源有限的设备上。例如，在一个没有网络通信能力的设备上，嵌入式Windows NT可省掉通用Windows NT中的网络管理工具、网卡和协议堆驱动程序等所有与网络相关的部分。

尽管如此，一些第三方厂商为Windows NT 4和Windows 2000/XP提供实时内核。它们将实时内核嵌在一个自定义的HAL中，把Windows 2000/XP或Windows NT 4当做实时系统中的一个任务来运行。这个Windows 2000和Windows NT任务可作为系统的用户接口，具有比设备管理任务低的优先级。

(3) 软件中断

虽然硬件产生了大多数的中断，但是Windows 2000/XP内核也为多种任务产生软件中断，它们包括：启动线程调度、处理定时器到时、在特定线程的描述表中异步执行一个过程以及支持异步I/O操作等。

2. 调度或延迟过程调用

当一个线程不能继续执行时，可能是由于它已经结束或者它进入了等待状态，内核直接调用调度程序将立即实现描述表切换。然而，有时内核在深入多层代码内时检测到应该进行重调度，在这种情况下，理想的解决方法是请求调度，延迟它的产生直到内核完成当前的活动为止。使用DPC软件中断是实现这种延迟的简便方法。

当需要同步访问共享的内核结构时，内核总是将处理器的IRQL提高到Dispatch/DPC级或高于Dispatch/DPC级，这样就禁用了其他的软件中断和线程调度。当内核检测到调度应该发生时，它将请求一个Dispatch/DPC级的中断；但由于IRQL等于或高于Dispatch/DPC级，处理器将在检查期间保存该中断。当内核完成当前活动后，它将IRQL降至低于Dispatch/DPC级，于是调度中断便可出现。通过使用软件中断来激活线程调度程序是延迟调度直到条件合适为止的一种方法。而Windows 2000/XP也使用软件中断来延迟其他类型的处理。

除了线程调度以外，内核在其他IRQL上也处理延迟过程调用。有一种DPC是执行系统任务的函数，该任务比当前任务次要。这些函数叫做“延迟函数”，因为它们可能不立即执行。DPC为操作系统提供了在内核态下产生中断并执行系统函数的能力。内核使用DPC处理定时器到时（并释放在定时器上等待的线程）和在线程时间片结束后重调度处理器。设备驱动程序使用DPC完成I/O请求。

DPC由DPC对象表示，它是一个内核控制对象。内核控制对象对于用户态的程序是不可见的，但对于设备驱动程序和其他系统代码是可见的。DPC对象包含的最重要的信息是当内核处理DPC中断时将调用的系统函数的地址。等待执行的DPC例程被保存在叫做“DPC队列”的内核管理队列中。为了请求一个DPC，系统代码将调用内核来初始化DPC对象，然后将它放入DPC队列中，如图2-10所示。

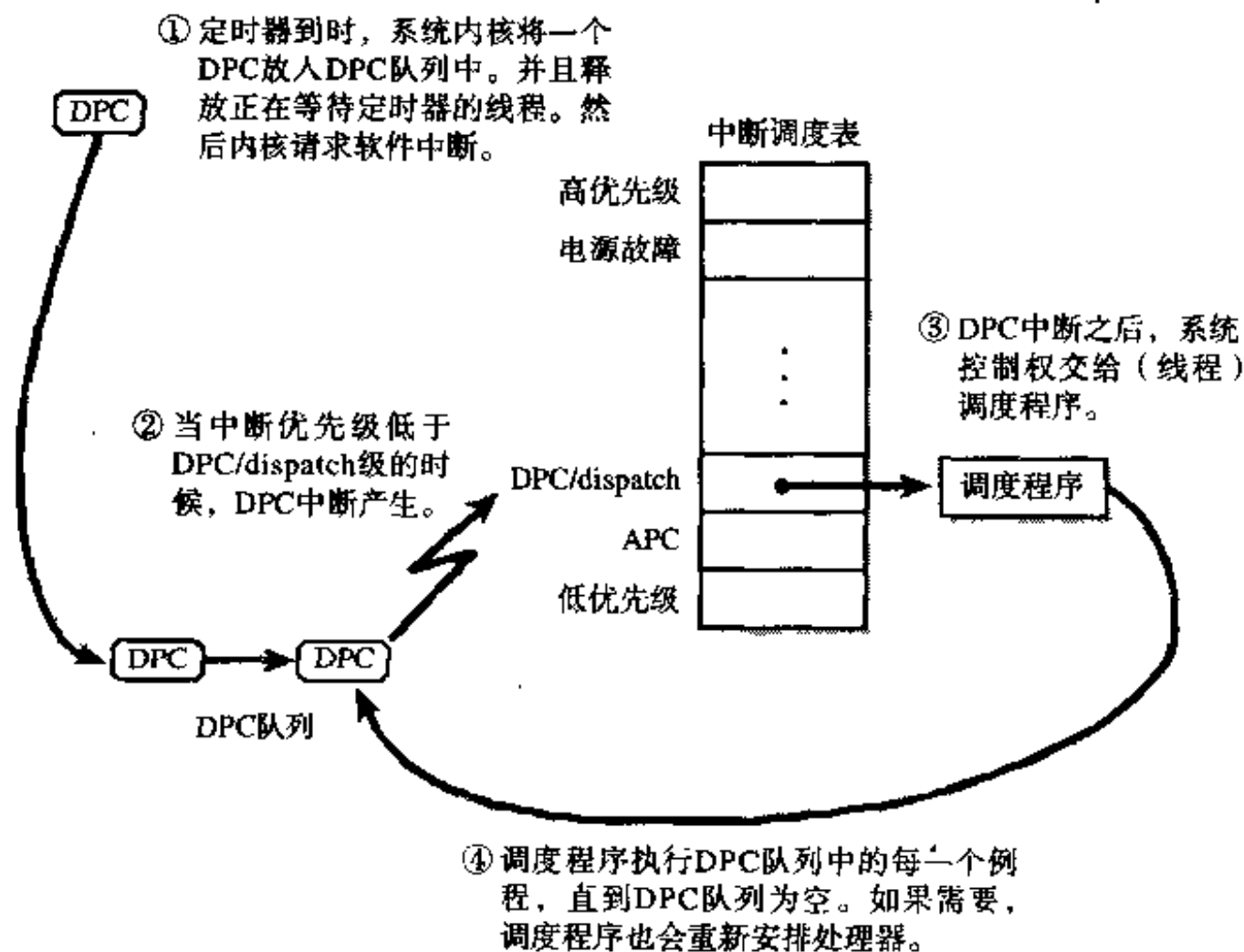


图2-10 延迟过程调用的提交

将一个DPC放入DPC队列会促使内核请求一个在Dispatch/DPC级的软件中断。因为通常DPC是由运行在较高IRQL级的软件对它进行排队的，所以被请求中断直到内核降低IRQL到APC级或

低于APC级时才出现。

用户态线程是以低IRQL执行的，这是DPC中断普通用户线程执行的良好时机。DPC例程执行不考虑什么线程正在运行，因而它不能假定当前映射的进程地址空间是什么。DPC例程可以调用内核函数，但不能调用系统服务、产生页面故障以及创建或等待对象。不过，它们可以访问非页面系统内存地址，因为不管当前是什么进程，系统地址空间总是可以被映射的。

DPC主要是为设备驱动程序提供的，但内核也使用它们，最经常的应用就是处理时间片到时。系统时钟的每个跳动在时钟IRQL都产生一个中断，时钟中断处理程序（运行在时钟IRQL）更新系统时间，并减小用来记录当前线程运行时间的计数器值。当计数器值到达零时，线程的时间片就已经到时，内核就可能需要重调度处理器，这是一个应该在Dispatch/DPC IRQL完成的低优先级的任务。时钟中断处理程序对DPC排队以启动线程调度，然后完成它的工作并降低处理器的IRQL。因为DPC中断的优先级低于设备中断的优先级，所以任何挂起的设备中断将在DPC中断产生之前得到处理。

3. 异步过程调用

异步过程调用（APC）为用户程序和系统代码提供了一种在特殊用户线程的描述表（一个特殊的进程地址空间）中执行代码的方法。APC在特殊线程描述表中执行，使用专用队列，它们以低于2的IRQL运行，所受的限制和DPC有很大的不同。APC例程可以获得资源（对象）、等待对象句柄、导致页错误以及调用系统服务。

APC也是由内核控制对象描述，称为APC对象。等待执行的APC在由内核管理的APC队列中。APC队列与DPC队列的不同在于：DPC队列是系统范围的；而APC队列是特定于线程的——每个线程都有自己的APC队列。当内核被要求对APC排队时，内核将APC插入到将要执行APC例程的线程的APC队列中。内核依次请求APC级的软件中断，并当线程最终开始运行时执行APC。

有两种APC，用户态APC和核心态APC。核心态APC在线程描述表中运行并不需要得到目标线程的“允许”，而用户态APC则需要得到目标线程的“允许”。核心态APC可以中断线程及执行过程，而不需要线程的干预或同意。

执行体使用核心态APC来执行必须在特定线程的地址空间（在描述表中）中完成的操作系统工作。例如，可以使用核心态APC命令一个线程停止执行可中断的系统服务，或记录在线程地址空间中的异步I/O操作的结果。环境子系统使用核心态的APC将线程挂起或终止自身的运行，或者得到或设置它的用户态执行描述表。POSIX子系统使用核心态APC来模仿POSIX信号到POSIX进程的发送。

设备驱动程序也使用核心态APC。例如，如果启动了一个I/O操作并且线程进入等待状态，则另一个进程中的另一个线程就可以被调度而去运行。当设备完或传输数据时，I/O系统必须以某种方式重新进入到启动I/O系统线程的描述表中，以便它能够来执行这个动作。

几个Win32 API，例如ReadFileEx、WriteFileEx和QueueUserAPC，使用用户态APC。例如，ReadFileEx和WriteFileEx函数允许调用者指定I/O操作完成时将被调用的完成例程。该完成例程是通过把APC排队到发出I/O操作的线程来实现的。然而，在对APC排队时，对完成例程的回调是没有必要的，因为仅当线程在“可报警等待状态”（alterable wait state）时，用户态APC才被

传送给线程。线程可以通过等待对象句柄并且指定它的等待是可报警的（使用Win32 WaitForMultipleObjectsE函数）进入等待状态，也可以通过直接测试它是否有一个挂起的APC（使用SleepEx）进入等待状态。在两种情况下，如果用户态APC是挂起的，内核将中断（报警）线程，将控制转交给APC例程，并在APC例程执行完成后，继续线程执行；如果发送APC的线程处于等待状态，在APC例程执行完成后，等待被重新发出或重新执行。如果等待仍没有解决，线程将返回到等待状态，但此时它将位于它正在等待的对象列表的末尾。

4. 异常调度

异常直接由运行程序的执行所产生，除了那些简单的可由陷阱处理程序解决的异常之外，所有异常都是由称作“异常调度程序”的内核模块提供服务。异常调度程序的工作是找到可以“处理”该异常的异常处理程序。由内核定义的与体系结构无关的异常例子包括内存访问越界、被零除、整数溢出、浮点异常和调试程序断点。Win32引入了称为“结构化异常处理”的工具，它允许应用程序在异常发生时可以得到控制。应用程序可以固定这个状态并返回到异常发生的地方展开堆栈，也可以向系统声明不能识别异常，并继续搜寻能处理异常的处理程序。

内核捕获和处理某些对用户程序透明的异常。内核通过给调用者返回不成功的状态代码来处理某些其他的异常。

少数异常可以被允许原封不动地过滤回用户态。环境子系统能够建立专用的异常处理程序来处理这些异常，它们与特殊过程的激活相关，称为基于框架的异常处理程序。当过程被调用时，表示该过程激活的堆栈框架就会被推入堆栈。堆栈框架可以有一个或多个与它相关的异常处理程序，每个处理程序都保护在源程序的一个特定代码块中。当异常发生时，内核将查找与当前堆栈框架相关的异常处理程序。如果没有，内核继续查找与前一个堆栈框架相关的异常处理程序，如此下去，直到找到一个基于框架的异常处理程序。如果还没有找到，内核将调用自己默认的异常处理程序。

异常调度的过程可见如图2-11所示的示意图。

异常会内核中会产生一个事件链，这与异常的来源无关。控制将转移到内核陷阱处理程序，陷阱处理程序将创建一个陷阱帧。如果完成了异常处理，陷阱帧将允许系统从中断处继续运行。陷阱处理程序同样将创建一个包含异常原因和其他有关信息的异常记录。

如果异常产生于核心态，异常调度程序将简单地调用一个例程来定位处理该异常的基于框架的异常处理程序。由于没有被处理的核心态异常将被视为致命的操作系统错误，因此可以假定调度程序总是能够找到异常处理程序。

程序调试断点也是异常的通常来源。异常调度程序的第一个动作就是查看引发异常的进程是否有相关的调试程序进程。如果有，它就给与引发异常的进程相关的调试程序端口发送第一个调试消息（通过本地过程调用LPC端口）。如果进程没有挂接调试程序进程，或者调试程序不处理异常，那么异常调度程序将切换到用户态，并调用例程来找到某个基于框架的异常处理程序。如果没找到任何基于框架的异常处理程序，或没有处理异常，异常调度程序就将切换回核心态，并再一次调用调试程序，以重新允许用户进行调试。

所有Win32线程都由处于堆栈顶的异常处理程序来处理未处理的异常。如果线程有一个没有

处理的异常，它就会调用Win32未处理异常筛选程序。如果调试程序未运行，并且没有找到基于框架的处理程序，内核就会给与线程的进程相关的异常端口发送一条消息。现存的异常端口是由控制这个线程的环境子系统登记的。异常端口给了假定正在该端口监听的环境子系统一个将异常转换为环境指定的信号或异常的机会。

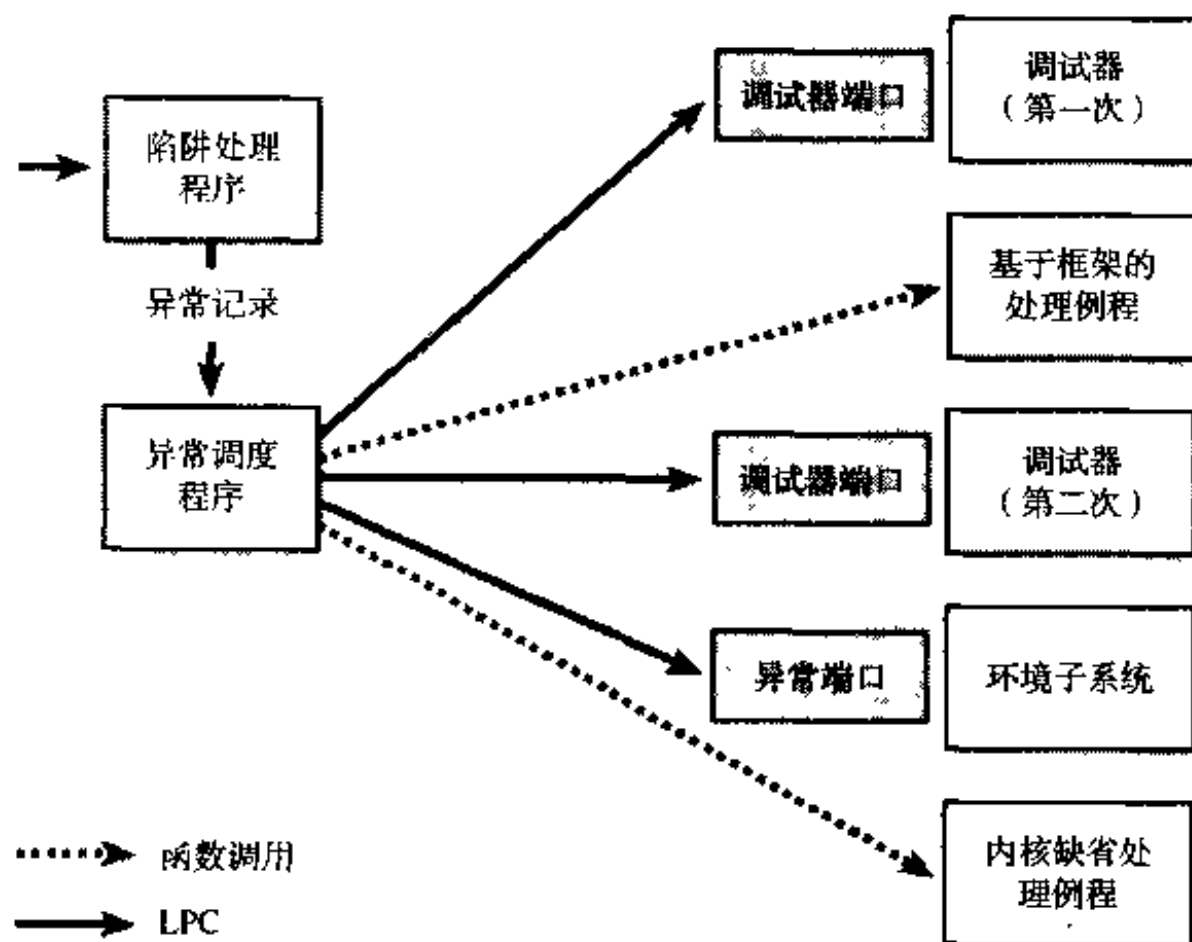


图2-11 Windows 2000/XP的异常调度

5. 系统服务调度

内核陷阱处理程序可以调度中断、异常以及系统服务调用。在Alpha处理器上执行syscall指令或在Intel x86处理器上执行int2E指令都会引起系统服务调度，这两个指令都可以使系统陷入系统服务调度程序。被传递的数值参数指明了被请求的系统服务号，内核使用这个参数查找位于“系统服务调度表”（system service dispatch table）中的系统服务信息。这个表和中断调度表相似，只是每个入口包含了一个指向系统服务的指针，而不是一个指向中断处理例程的指针。

系统服务调度程序将校验参数，并将调用者的参数从线程的用户堆栈复制到它的核心堆栈中，然后再执行系统服务。如果传递给系统服务的参数指向用户空间的缓冲区，则核心态代码在访问缓冲区之前，会查明这些缓冲区的可访问性。

每个线程都有一个指向系统服务表的指针。Windows 2000/XP有两个内置的系统服务表：第一个默认表定义了NTOSKRNL.EXE中实现的核心执行体系统服务；另一个表是在Win32子系统Win32K.SYS的核心态部分中实现的Win32 USER及GDI。当Win32线程第一次调用Win32 USER或GDI时，线程系统服务表的地址将指向包含Win32 USER和GDI的服务表。

用于执行体服务的系统服务调度指令存于系统库NTDLL.DLL中。子系统动态链接库通过调用NTDLL中的函数来实现它们的文档化函数。这里有一个例外是Win32 USER及GDI，出于效率

的考虑，其中的系统服务调度指令是在USER32.DLL和GDI32.DLL中直接实现的——中间没有NTDLL.DLL。如图2-12所示。

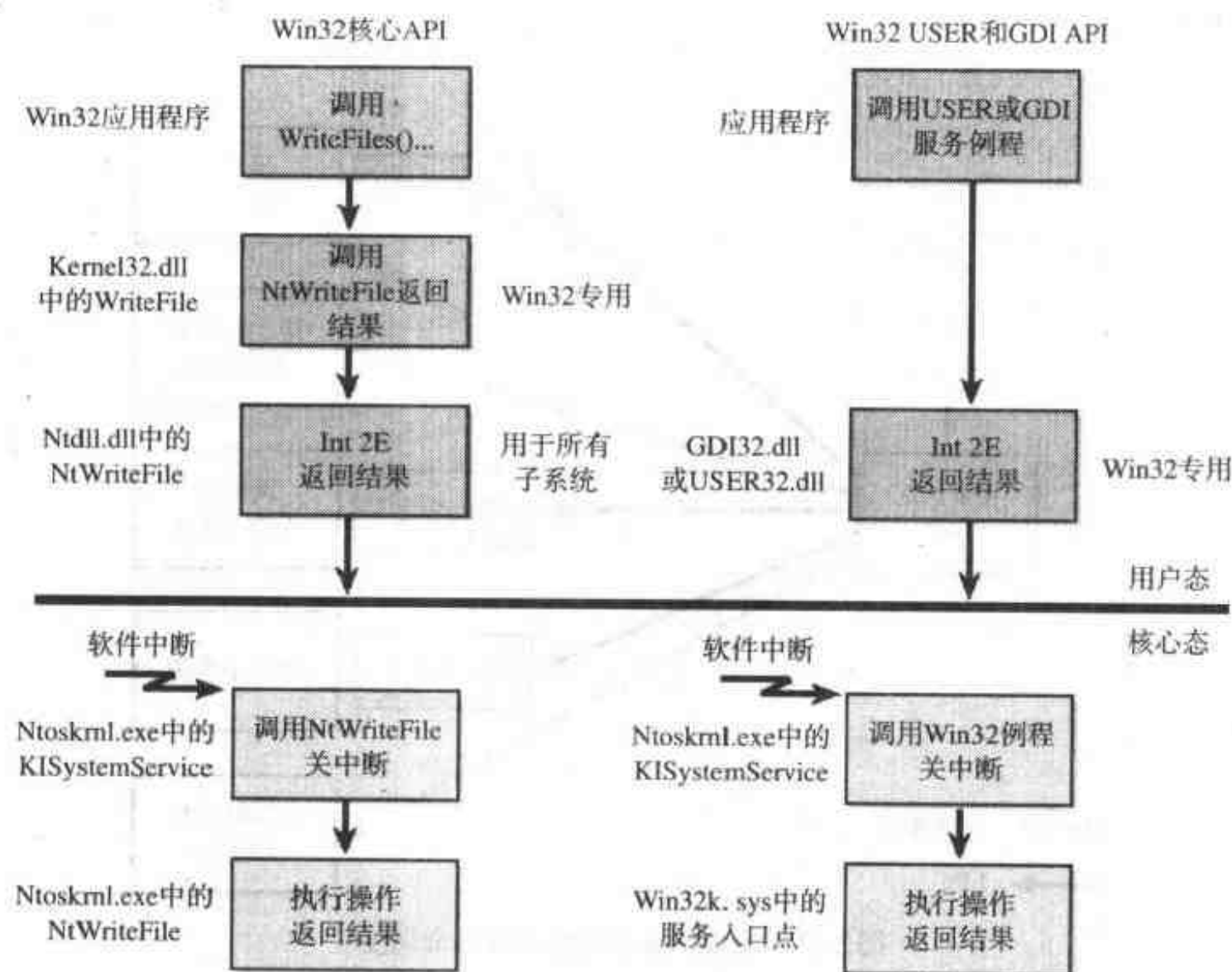


图2-12 系统服务调度

2.4.2 对象管理器

Windows 2000/XP通过对象管理机制为执行体中的各种内部服务提供一致的和安全的访问手段。对象管理器是一个用于创建、删除、保护和跟踪对象的执行体组件。对象管理器提供使用系统资源的公共、一致的机制；把对象保护孤立到操作系统的一个位置，实现安全保护。它提供了一种机制来控制进程使用对象，实现了资源的访问控制。对象管理器有一套对象命名方案和统一的保留规则，能够容易地操纵现有对象。同时，对象管理器能支持各种操作系统环境的需要。

Windows 2000/XP中有两种类型的对象：执行体对象和内核对象。执行体对象就是由执行体的各种组件实现的对象（例如进程管理器、内存管理器、I/O子系统等）。内核对象是由内核实现的一个更原始的对象集合，这些对象对用户态代码是不可见的，它们仅在执行体内创建和使用。内核对象提供了一些基本性能，许多执行体对象内包含着一个或多个内核对象。

执行体对象和对象服务都是基本设施，环境子系统用它们来构造自己版本的对象和资源。执行体对象典型地由代表某个用户应用程序的环境子系统创建，或者由操作系统的不同组件作为它

们正常操作的一部分来创建。环境子系统为其应用程序提供的对象集一般与执行体所提供的对象集有些差异。Win32子系统使用执行体对象导出它自己的对象集，其中的大部分对象直接符合执行体对象。

每一个对象都有一个对象头和一个对象体。对象管理器控制对象头，各执行体组件控制它们自己创建的对象类型的对象体。另外，每一个对象头都指向打开该对象的进程的列表，同时还有一个叫做类型对象的特殊对象，它包含的信息对每一个对象的实例是公用的。

1. 对象头

对象管理器利用存储在对象头中的数据来管理对象，对象体的格式和内容对于它的对象类型来说则是唯一的。通过创建类型对象并为其提供服务，执行体组件可以控制这个类型的所有对象体的数据处理。对象管理器提供一个小的通用服务集用于操作对象头中存储的属性，不过基于效率和实现复杂度的考虑，每个对象都单独实现自己的创建、打开和查询服务。

2. 类型对象

对象头包含的数据对所有的对象是公用的，但是对于对象的每个实例，其数据可以取不同的值。例如，每个对象都有唯一的名称，并且可以有唯一的安全描述体。对象同样包含对特定类型的所有对象都是常数的一些数据。为了节省内存，对象管理器在创建一个新的对象类型时，就存储这些静态的指定对象类型的属性。它使用自己的叫做类型对象的对象来记录这些数据。因为对象管理器没有为它们提供服务，所以类型对象是不能从用户态直接进行处理的，不过，它们定义的一些属性通过某些本机服务和Win32 API程序则是可见的。

同步是指一个线程通过等待一个对象从一种状态转变为另一种状态来同步执行的能力。线程可以和执行体进程、线程、文件、事件、信号量、互斥体以及定时器对象同步。区域、端口、访问令牌、对象目录、符号链接、配置文件和键对象不支持同步。

3. 对象方法

方法包含与C++构造函数和析构函数类似的一组内部例程。对象管理器通过在另外一些情况下调用对象方法延伸了这个思想，有些对象类型指定方法，有些则不指定，这依赖于对象类型是如何被使用的。

当一个执行体组件创建一个新的对象类型时，它可以向对象管理器注册一个或多个方法。此后，对象管理器在那个类型对象的生命周期中预先定义好的点上可以调用这些方法，通常是在对象以某种方式被创建、删除或做某些修改时调用它们。

一个关闭方法的使用例子发生在I/O系统中。I/O管理器为文件对象类型注册了一个关闭方法，这样对象管理器在每次关闭文件对象句柄时都将调用此关闭方法。这个关闭方法检查正在关闭文件句柄的进程中是否有任何没有打开的文件锁，如果有，就把它移走。检查文件锁并不是对象管理器自己能够或应该做的。如果一个删除方法已经被注册，对象管理器在从内存中删除一个临时对象前会调用该删除方法。如果二级对象管理器找到一个存在于对象管理器名字空间之外的对象时，分析方法允许对象管理器把寻找对象的控制下放给一个二级对象管理器。当对象管理器寻找一个对象名时，一旦它遇到了有相应分析方法的对象，它就挂起搜索。对象管理器调用此分析方法，把正在搜索对象名的其余部分传递给它。

I/O系统使用的安全方法与分析方法相类似。无论何时线程要改变保护文件的安全信息时，就调用安全方法。文件的安全信息与其他对象的安全信息不同，因为安全信息被存储在文件自身而不是存储在内存中。因此，必须调用I/O系统来发现安全信息并改变它。

4. 对象句柄和进程句柄表

当进程通过名称来创建或打开一个对象时，它会收到一个代表进程访问对象的句柄。通过句柄指向对象比使用名称要快，因为对象管理器可以跳过名称搜索而直接找到对象。进程也可以通过在进程创建时继承句柄或从其他的进程接收复制句柄来为对象获得句柄。

所有用户态进程只有获得了对象句柄之后才可以使用这个对象。句柄作为系统资源的间接指针来使用，这种不直接的方式阻止了应用程序对系统数据结构直接地随便操作。

对象句柄提供另外的一些好处。首先，除了它们引用了什么以外，在文件句柄、事件句柄和进程句柄之间并没有不同。这种相似性为引用对象提供了统一的界面而忽略它们的类型。其次，对象管理器有创建句柄和定位句柄引用对象的专用权限。这就意味着对象管理器能够细察影响对象的每个用户态的操作，以检查调用者的安全配置文件是否允许在该对象上执行所请求的操作。

对象句柄是一个由执行体进程EPROCESS所指向的进入进程句柄表的索引。进程句柄表包含进程已为其打开句柄的所有对象的指针。它由一个固定的表头和一个大小可变的部分组成。大小可变的部分是一个句柄表入口数组，每一项描述了一个打开的句柄。如果一个进程打开了许多句柄，而且比可变部分能容纳的句柄多，则系统就将重新分配一个新的、更大的数组，并把旧数组复制到新的数组中。

每个句柄入口由一个包含两个32位成员的结构组成。第一个32位成员包含一个指向对象头的指针和三个标志。第一个标志是继承标志——即由这个进程创建的一些进程是否将在它们的句柄表中得到句柄的副本。第二个标志指出是否允许调用者关闭这个句柄。第三个标志指出在关闭对象时是否将生成一个审计消息。第二个32位成员是用于那个对象的已授权的访问掩码。

5. 对象安全

当进程创建对象或打开一个现存对象的句柄时必须指定一组期望的访问权限，此时对象管理器将调用安全引用监视程序（安全系统的核心态部分），并把进程的一组期望的访问权限集发送给它。安全引用监视程序将检查对象的安全描述体是否允许进程正在请求的访问类型。如果允许，安全引用监视程序将返回允许进程访问的一组授予访问权限，对象管理器将把它们存储在它创建的对象句柄中。

此后，无论何时当进程的线程使用句柄时，对象管理器都可以快速地检查在句柄中存储的已授权访问权限集是否符合由线程调用的对象服务隐含的用法。例如，如果调用者请求读取访问某个区域对象，但是在这之后他调用了—个对区域对象的写入服务，那么写入就会失败。

6. 对象保留

因为所有访问对象的用户态进程必须首先为它打开一个句柄，所以对象管理器可以很容易地跟踪有多少进程，甚至是哪些进程正在使用该对象。跟踪这些句柄代表了实现对象保留的第一步——即只在使用对象时临时保留对象，然后删除它们。

对象管理器分两个阶段实现对象保留。第一个阶段叫做“名称保留”，它是由对象的打开句

柄数目来控制的。每次进程为对象打开句柄时，对象管理器就增加对象头中的打开句柄计数器值。当进程使用完对象并且关闭它的句柄时，对象管理器就减少打开句柄计数器值。当计数器减少到零时，对象管理器就从全局名字空间中删除对象名。这个删除防止了新进程为这个对象打开句柄。对象保留的第二阶段是当对象不再被使用时，停止保留它们。因为操作系统代码经常使用指针代替句柄访问对象，所以对象管理器必须同样记录有多少对象指针已经分配给操作系统进程。每次对象管理器给对象分配一个指针，它就为此增加一个引用计数；当核心态组件结束使用该指针时，调用对象管理器减少对象的引用计数。所以，即使在一个对象的打开句柄计数器值为零以后，该对象的引用计数仍然可能是正的，这就表明操作系统仍然在使用该对象。最终，引用计数也会降为0。当这种情况发生时，对象管理就从内存中删除该对象。

根据对象保留的工作方法，只要简单地对象保持一个打开它的句柄，应用程序就能确保对象和它的句柄仍然保留在内存之中。

7. 资源记账

资源记账像对象保留一样，与使用对象句柄密切相关。资源对象打开的句柄计数为正，表明某个进程正在使用其资源，它同样指明该进程由于对象占用内存而正被记入账内。

许多操作系统使用一个配额系统来限制进程对系统资源的访问。然而，为进程分配的配额类型有时是多样的、复杂的，并且跟踪配额的代码分布在操作系统内各处。Windows 2000/XP的对象管理器为资源记账提供中心服务。每一个对象头包含一个称为配额账的属性来记录当进程中的一个线程为对象打开句柄时对象管理器从分配给进程的页交换区和/或非页交换区配额中减去了多少。

在Windows 2000/XP上的每个进程都指向一个配额结构，它记录了用于非页交换区/页交换区和页面文件用法的限制和当前值。然而，在您的交互式会话中，所有进程共享同一个配额块，系统进程没有配额限制。

页交换区的配额大小从521KB开始，非页交换区的配额大小从64KB开始。这种限制是“柔性的”，当进程超出配额限制时，系统会试图自动增加进程配额。如果打开一个对象将超出页交换区或非页交换区配额，系统就将调用内存管理器看看能否增加配额，如果不能，那么打开对象的请求就会失败，并将给出“超出配额”的错误。但是在大多数系统中，配额会继续根据需要而增加。

8. 对象名

创建大批对象的一个重要考虑是如何跟踪它们。对象管理器提供了把某个对象与其他对象区别的方法以及查找并检索一个特定对象的方法。

这两点通过给对象命名实现，执行体允许任何由对象所代表的资源有一个名称。如果对象管理器用名称存储对象，就能通过寻找名称来查找对象。对象名还使得对象可以在不同进程间共享。执行体对象名字空间是全局性的，对系统中所有过程可见。一个进程可以创建一个对象并将它的名称放入全局名字空间，而且第二个进程可以通过指定对象名称来为对象打开句柄。如果对象不想以这种方式共享，它的创建者不需要给它命名。

为了提高效率，对象管理器并非在每次有人使用对象时就检索对象名，而只在两种情况下寻

找名称，第一种情况是当进程创建一个名称的对象时，对象管理器要在名字空间中寻找这个名称以证实它还没有存在；第二种情况是当进程为一个命名的对象打开句柄时，对象管理器寻找名称并找到对象，然后给调用者返回一个对象句柄。

基本的核心对象（如互斥体、事件、信号量、可等待定时器和区域）将各自的名称存储在同一个对象目录中，它们是不能重名的。对象名对单个计算机来说是全局性的，但它们在网络上是不可见的。

对象目录对象是对象管理器支持这种层次命名结构的方法。这个对象与文件系统目录类似，并且包含其他对象，甚至可能是对象目录的名称。对象目录对象保持了足够的信息将这些对象名称转化成指向对象自身的指针。

在某些文件系统中，符号链接使用户能创建一个文件名或目录名，而当使用时，它被操作系统转化成不同的文件或目录名。使用符号链接是一个简单的方法，它允许用户间接地共享一个文件或目录的内容，来用的是在通常层次目录结构中不同目录之间创建交叉链接的方法。对象管理器实现一个叫做符号链接对象的对象，符号链接对象为在它的对象名空间内的对象名执行一个相似的功能。符号链接可以在对象名字字符串中的任何地方发生。执行体使用符号链接对象的一个场合是把MS-DOS中的设备名转换成Windows 2000/XP的内部设备名。此外，用户可以使用subst命令添加伪驱动器名或映射一个驱动器名到网络共享。一旦它们被创建，对于在系统内的所有进程，它们都必须是可见的。

2.4.3 同步

互斥指的是保证有一个、并且每次只有一个线程可以访问一个特殊的资源。当资源不支持它自身被共享访问或当共享时会导致意外的输出时，互斥是必要的。互斥对于紧耦合的支持对称多处理的操作系统来说尤其重要，在这些操作系统中，相同的系统代码能同时在多个处理器上运行，共享存储在共用内存中的某些数据。在Windows 2000/XP中，内核的工作是提供一种机制，它可以使用系统代码来防止两个线程同时修改同一个结构。内核提供简单的互斥，它和执行体的其余部分用于同步对共用数据结构的访问。

1. 内核同步

在内核执行的不同阶段，它必须保证有一个、并且每次只有一个处理器在临界区执行。内核的临界区是修改共用数据结构的代码段，例如内核的调度程序数据库或它的DPC队列。只有在内核能保证线程以互斥方式访问这些数据结构时，操作系统才能正常工作。

所涉及的最大问题是中断。当中断产生时，它的中断处理例程会修改公共数据结构，而此时内核也许正在更新这个共用数据结构。Windows 2000/XP内核在使用共用资源以前，将暂时屏蔽那些其中断处理程序也要使用该资源的中断。它通过把处理器的“中断请求级”（IRQL）提高到由任意潜在的访问全局数据的中断资源使用的最高级来达到这个目的。

这种策略对于单处理器系统是很好的，但不适用于多处理器结构。内核用自旋锁实现了多处理器互斥机制。自旋锁是一个与共用数据结构有关的锁定机制，如图2-13。

在进入图2-13中所示的两个临界区中的一个之前，内核必须获得与所保护的DPC队列有关的

自旋锁。如果自旋锁非空，内核将一直尝试得到锁直到成功。因为内核被保持在过渡状态“旋转”，直到它获得锁，所以自旋锁由此而得名。

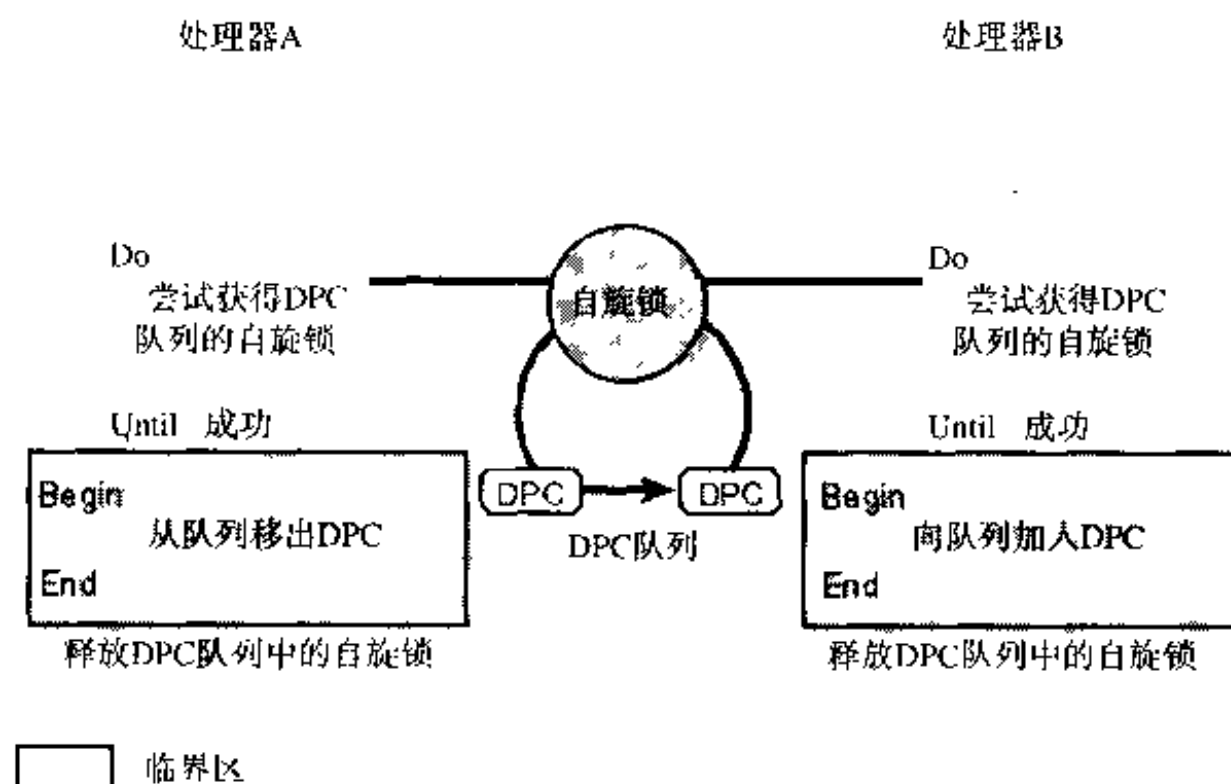


图2-13 自旋锁的使用

自旋锁像它们所保护的数据结构一样，存储在共用内存中。为了提高速度和使用任何在处理器体系下提供的锁定机构，获取和释放自旋锁的代码是用汇编语言写成的。在很多体系结构中，自旋锁是使用一个硬件支持的测试并设定单一指令来实现的，它测试锁变量值并在一条基本指令中获得锁。在一条指令中测试和获得锁，防止了当第一个线程在测试变量到获得锁的这一时间段内第二个线程夺取锁。

当线程试图获得自旋锁时，在处理器上的所有其他工作将终止。因此，拥有自旋锁的线程永远不会被抢先，但允许它继续执行以使它尽快把锁释放。内核对于使用自旋锁十分小心，当它拥有自旋锁时，它执行的指令数将减到最少。

通过一组内核函数，内核使执行体的其他部分也可以使用自旋锁。

2. 执行体同步

在多处理器环境下，在内核之外的执行体软件同样需要同步访问共用数据结构。自旋锁只是部分满足了执行体对同步机制的需要，这是因为在自旋锁上等待，实际上是使处理器暂停，自旋锁只可以在下列严格受限的环境使用：

- 被保护的资源必须被快速访问，并且没有与其他代码的复杂的交互作用。
- 临界区代码不能换出内存，不能引用可分页数据，不能调用外部程序（包括系统服务），不能生成中断或异常情况。

内核以内核对象的形式给执行体提供额外的同步机构，统称为调度程序对象。Win32应用程序中的一个线程可以与一个Win32的进程、线程、事件、信号量、互斥体、可等待定时器、I/O完成接口或文件对象同步。执行体同步对象的类型叫做执行体资源，它实现了独占访问（如互斥体）以及共享只读访问。但是，它们仅对核心态代码是可以使用的，而不能从Win32 API访问。资源

不是调度程序对象，而是从非页交换区直接分配的数据结构，它们有自己的专门服务来初始化、锁定、释放、查询和等待。

(1) 等待调度程序对象

一个线程通过等待对象的句柄可以与调度程序对象同步，这样做会使内核将线程挂起并把它调度状态从运行改为等待。在任何给定时刻，同步对象会处于“有信号状态”或“无信号状态”。一个线程在内核将其调度状态由等待状态改为就绪状态时才能恢复它的执行。当线程正在等待其句柄的调度程序对象也在经历一次状态改变时，这种改变就会发生。线程调用由对象管理器提供的等待系统服务程序来与对象同步，给希望同步的对象传递句柄。线程可以等待一个或几个对象，如果等待在一定时间内没有结束，线程也可以指定取消它的等待。无论什么时候当内核将一个对象设置为有信号状态时，它将检查是否有线程在等待这个对象。如果有，内核会把一个或几个线程从它们的等待状态释放，这样它们就能继续执行。

下面设置事件的例子说明了同步是如何与线程调度互相作用的（如图2-14）。

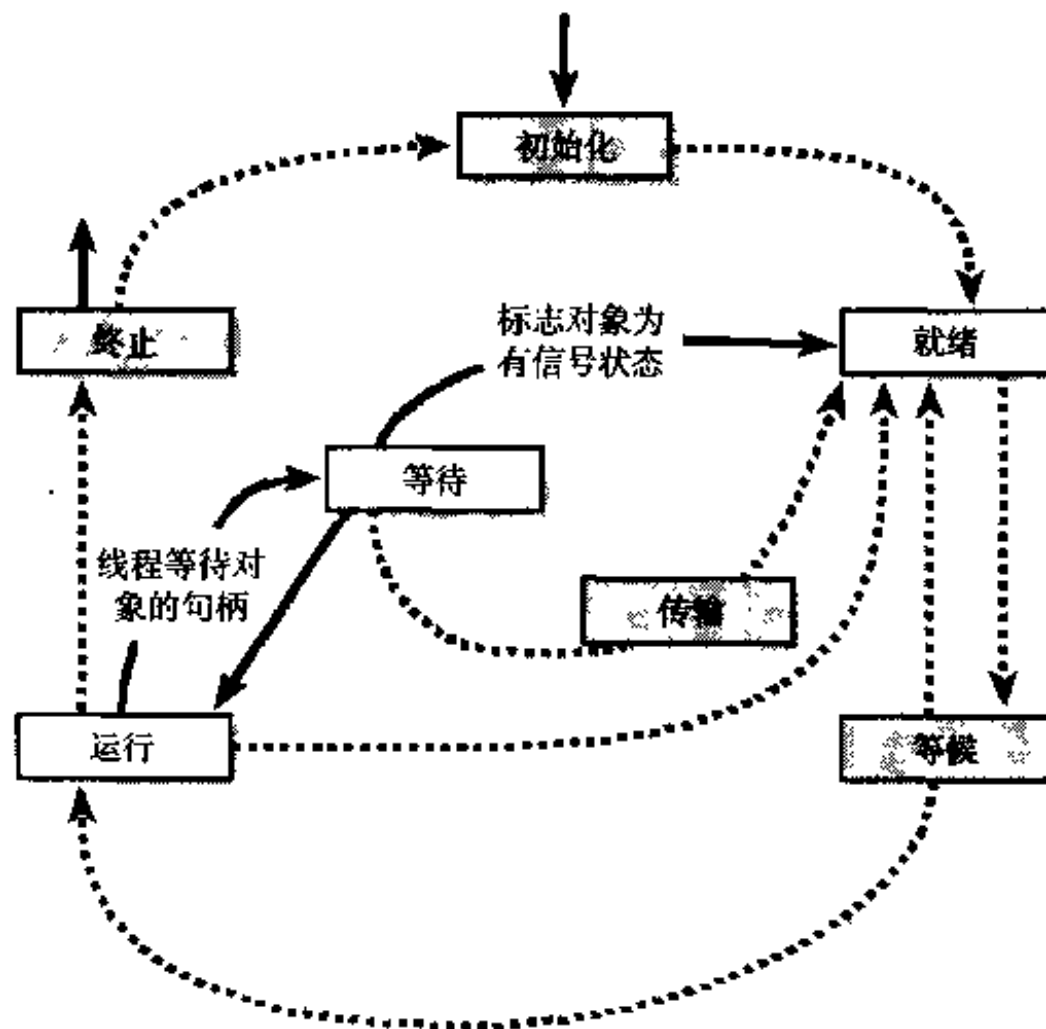


图2-14 等待一个调度程序对象

- 1) 用户态线程等待事件对象句柄。
- 2) 内核把线程的调度状态由就绪改为等待，然后将线程添加到等待事件的线程列表中。
- 3) 另一个线程设置事件。
- 4) 内核从等待事件的线程列表向下匹配。如果满足了线程的等待条件，内核就把线程的状态由等待改为就绪。如果是可变优先级的线程，内核也可能提高它的执行优先级。
- 5) 因为新线程已经成为就绪状态而去执行，所以调度程序将重新安排线程。如果它发现正在

运行线程的优先级比新的就绪线程的优先级低时，它就会抢先低优先级线程，并发布一个软件中断为高优先级的线程初始化描述表切换。

6) 如果没有处理器可以被抢先，调度程序将把就绪线程放入调度程序就绪队列中，以备稍后调度。

(2) 何种情况把对象置为有信号状态

对于不同对象，有信号状态的定义不同。在线程对象的生存期内，它处于无信号状态；当线程终止时，内核把它设置为有信号状态。类似地，当进程的最后一个线程终止时，内核把进程对象设置为有信号状态。与之相反，定时器对象（像闹钟一样）在一个确定的时间被设置为“发声”，当它的时间期满以后，内核设置定时器对象为有信号状态。

在选择同步机制时，程序必须考虑管理不同的同步对象行为的规则。当对象被设置为有信号状态时，线程的等待是否结束取决于线程正在等待的对象类型。当一个对象被设置为有信号状态时，处于等待状态的线程通常被立刻释放。

通知事件对象被用来宣布某些事件的发生。当事件对象被设置为有信号状态时，将释放所有正等待该事件的线程；但当一个线程每次等待的事件多于一件时除外。在另外的对象达到有信号状态以前，这样的线程可能将继续等待。

与事件对象相反，互斥体有与它自身相关的所有权。它被用来获得对某个资源的互斥访问，并且每次只有一个线程能拥有互斥体。当互斥体对象空闲，内核就将它设置为有信号状态并选择一个正在等待的线程去执行，被内核先中的线程将获得互斥体对象，而所有其他的线程都将继续等待。

(3) 数据结构

对于跟踪谁在等待什么，有两个数据结构是很关键的：调度程序头和等待块。在DDK的包含文件ntddk.h中的这两种结构是公开的。

```
typedef struct _DISPATCHER_HEADER
{
    UCHAR Type;
    UCHAR Absolute;
    UCHAR Size;
    UCHAR Inserted;
    LONG SignalState;
    LIST_ENTRY WaitListHead;
} DISPATCHER_HEADER;

typedef struct _KWAIT_BLOCK
{
    LIST_ENTRY WaitListEntry;
    struct _KTHREAD *RESTRICTED_POINTER Thread;
    PVOID object;
    struct _KWAIT_BLOCK *RESTRICTED_POINTER NextWaitBlock;
    USHORT waitKey;
    USHORT waitType;
} KWAIT_BLOCK, *PKWAIT_BLOCK, *RESTRICTED_POINTER PKWAIT_BLOCK;
```


调度程序头包含了对类型、有信号状态和正在等待对象线程的列表。等待块代表了正在等待对象的线程。每个处于等待状态的线程都有等待块列表，它代表了线程正在等待的对象。每个调度程序对象都有一个等待块列表，代表正在等待该对象的是哪些线程。这个列表将被一直保存，这样当调度程序对象处于有信号状态时，内核就能迅速判断谁正在等待这个对象。等待块有一个指向被等待对象的指针、一个指向等待该对象的线程指针和一个指向下一个等待块的指针。指针也记录等待类型以及在句柄数组中入口的位置，该位置是由线程在调用WaitForMultipleObjects函数时传送的（若线程只等待一个对象时为0）。

2.4.4 本地过程调用

本地过程调用（LPC）是一个用于高速信息传输的进程间通信机构，在Win32 API下它是不可用的；它是一个对Windows 2000/XP操作系统组件有效的内部机制。下面是一些使用LPC的例子：

- 远程过程调用使用LPC在同一个系统中的进程之间通信。
- 少数Win32API导致给Win32子系统进程发送信息。
- WinLogon使用LPC与LSASS通信。
- 安全引用监视器，使用LPC与LSASS进程通信。

典型地，在一个服务器进程与该服务器的一个或多个客户进程之间的两个用户态进程之间或一个核心态组件和一个用户态进程之间可以建立LPC连接。

LPC被设计成允许三种交换信息的方法：

1) 使用包含信息的缓冲区调用LPC可以发送少于256字节的信息。然后，这个信息又从发送进程的地址空间复制到系统地址空间，再从那里拷贝到接收进程的地址空间。

2) 如果用户和服务端想交换大于256字节的数据，那么他们可以选择使用双方都映射了的共享区。发送方将信息数据放到共享区，然后向接收方发送一小段信息表明在共享区的什么地方可以找到数据。

3) 当服务器想读或写大量数据，而共享区又太小时，数据可以直接从客户地址空间读出或向客户地址空间写入。LPC组件提供了两个函数，服务器可以用它们来完成这些操作。以第一种方式发送的信息被用于同步正在传送的信息。

LPC导出一个称为端口对象的单个执行体对象，用它来保持通信所需要的状态。尽管LPC使用单个对象类型，但是它有几个端口：① 服务器连接端口是一个已命名的服务器连接请求指向端口，客户可通过与这个端口连接从而连接到服务器上；② 服务器通信端口是一个未命名的用于特殊通信的端口，服务器与每一个活动客户都有一个这样的端口；③ 客户通信端口是特殊客户线程用来与特殊服务器通信的未命名的端口；④ 未命名通信端口是为用于同一进程中的两个线程而创建的未命名的端口。

LPC典型地应用于以下情况：服务器创建已命名的服务器连接端口对象。客户提出与这个端口连接的请求。如果同意该请求，客户通信端口和服务器通信端口就会被创建。客户得到了客户通信端口的句柄，服务器得到服务器通信端口的句柄。然后客户和服务端将为了它们之间的通信而使用一些新的端口。

2.4.5 系统工作线程

在系统初始化的过程中，Windows 2000/XP 在系统进程中产生了很多线程，成为系统工作线程，它们独立的存在，代表其他线程履行职责。很多时候，线程运行在延迟过程调用或者调度级别时需要调用在较低中断请求级别才能执行的功能，而当它们不能降低中断请求级别的时候，它们就要将处理权交给一个运行在更低中断请求级别的工作线程。

有些设备驱动程序和执行体组件产生它们自己的线程用于在被动级别处理工作；不过，大多数设备驱动程序和执行体组件使用系统工作线程，因为它们避免了由于增加额外系统线程而造成的不必要的调度和内存开销。设备驱动程序和执行体组件通过调用ExQueueWorkItem和IoQueueWorkItem请求一个系统工作线程。这些功能放置一个工作项在工作线程寻找工作的队列调度器对象上。工作项包括一个指向例程的指针和工作线程处理这些工作项时需要的参数。这些例程由相应的请求被动级别执行的设备驱动程序和执行体组件实现。

有三种系统工作线程：① 延迟工作线程运行在优先级12，处理非时间关键的工作项，它们的堆栈在等待工作项时可以被换出到页交换文件；② 关键工作线程运行在优先级13，处理时间关键的工作项，在Windows2000 server中它们的堆栈始终在物理内存中；③ 一个单独的高度关键的工作线程运行在优先级15，堆栈也始终在物理内存中，处理管理器使用这种工作线程的“收割机”功能释放终止的线程。

2.5 Windows 2000/XP的注册表

注册表在配置和管理Windows 2000系统的机制中扮演着关键的角色。它存储着所有关于系统和每个用户的设置信息。

2.5.1 注册表的数据类型

注册表是一个数据库，它的结构与磁盘的逻辑结构相似。注册表包括主键和键值，它们之间的关系就像磁盘上的目录和文件一样。一个主键可以包含若干个主键（或称为这个主键的子键）和键值，而键值则存储数据，顶级主键称为根键。

主键和键值借用了文件系统的命名习惯。每个键值都有一个名称，我们可以用这个名称唯一确定一个键值。在这个命名机制中，唯一的例外是未命名的主键，我们可以从两个不同的注册表编辑器中看出其中区别，regedit用<default>来代表未命名的键值，而regedt32则用<no name>来表示未命名的键值。

键值可以存储不同类型的数据。注册表中大多数的键值是REG_DWORD、REG_BINARY或REG_SZ类型的。REG_DWORD存储数值或布尔类型的数据；REG_BINARY可以存储任意长度的二进制数以及一些原始数据，如加密的口令；REG_BINARY存储Unicode的字符串，比如姓名、文件名、路径和类型。

REG_LINK类型比较特别，它可以让一个主键显式地指向另一个主键。当访问这种类型的主键时，系统会沿着链接直到目的地。Windows 2000/XP在很多情况下都用到了指针类型的键值，

在6个根主键中有3个根主键是指向另3个非指针类型根主键的子键。指针类型的主键不会被存储，它们必须在每次系统启动时被动态地创建。

2.5.2 注册表的逻辑结构

注册表有6个根主键，这些根主键不可以被删除，也不能添加新的根主键，它们存储的信息如下：

- **HKEY_CURRENT_USER** 存储与当前登录用户有关的信息。
- **HKEY_USER** 存储了所有用户的信息。
- **HKEY_CLASSES_ROOT** 存储与文件类型和COM对象相关的信息。
- **HKEY_LOCAL_MACHINE** 存储与系统设置相关的信息。
- **HKEY_PERFORMANCE_DATA** 存储与系统性能相关的信息。
- **HKEY_CURRENT_CONFIG** 存储了当前硬件配置文件的信息。

1. HKEY_CURRENT_USER

HKCU 包含了与当前登录用户相关的软件配置和参数。它对应当前用户的用户配置文件，这个文件在磁盘上的位置是：“\Documents and Settings\<username>\ntuser.dat”。当系统载入用户配置文件（比如登录或这个用户调用一个服务进程时），系统将创建一个指向**HKEY_USERS**下代表该用户的子键的**HKCU**类型的链接。

2. HKEY_USERS

它包含了为每一个用户配置文件所创建的子键以及用户在系统数据库中注册的类信息。它还包括一个叫做“**HKU\DEFAULT**”的子键，这个子键指向缺省的工作站配置文件。

3. HKEY_CLASSES_ROOT

这个主键包括文件扩展名和COM组件类的注册信息。大多数的主键都包含一个**REG_SZ**类型的键值，它指向**HKCR**中与存储这种文件扩展名信息相关的另一个主键。比如，“**HKCR\.xls**”指向与Microsoft Excel文件信息相关的主键（如“**HKCU\Excel.Sheet.8**”）。其他主键包含了那些已在系统里注册过的COM组件的配置信息。

HKEY_CLASSES_ROOT下的数据来自两个方面：

- **HKCU\SOFTWARE\Classes**包括了用户的注册类（对应的磁盘文件映射是“\Documents and Settings\<username>\Local Settings\Application Data\Microsoft\Windows\Usrclass.dat”）。
- **HKLM\SOFTWARE\Classes**包括了系统的注册类。

4. HKEY_LOCAL_MACHINE

HKLM根主键包含了所有有关整个系统的配置信息的子键：**HARDWARE**、**SAM**、**SECURITY**、**SOFTWARE**和**SYSTEM**。

HKLM\HARDWARE子键包含系统硬件配置脚本和所有设备驱动程序的映像。设备管理工具所提供的系统的硬件配置信息就是直接从**HARDWARE**这个主键中读出的。

HKLM\SAM子键保存本地的用户和组信息，如用户密码、组和域的定义。和域控制器相似，Windows 2000/XP Server将域和组的用户信息存储在活动目录里，这个活动目录实际是一个存储

域级设置和信息的数据库。在缺省状态下，SAM主键的安全级别会设为对任何用户（包括系统管理员）都不可访问，如果你想要了解这个主键的内容，你可以将这个主键的安全级别修改为对管理员可读，但你可能得不到什么有用的信息。

HKLM\SECURITY子键存储了系统的安全策略以及用户的权限设置。上面所提到的HKLM\SAM事实上是链接到HKLM\SECURITY\SAM下的SECURITY子键的。在缺省情况下，你是不可能看到HKLM\SECURITY或 HKLM\SECURITY\SAM的内容的，因为它们的安全级别被设为系统级用户才可以访问。

HKLM\SOFTWARE子键存储了一些Windows 2000系统级软件配置信息，但通常要使对这些信息的更改生效是不用重新启动系统的。这个子键还存储了第三方应用程序的系统级设置，如应用程序中文件和目录的路径，以及有关使用许可证和期限的信息。

HKLM\SYSTEM子键同样存储了一些系统级的配置信息，但我们必须重新启动系统才可以使对这些信息的更改生效，比如加载哪些设备驱动程序或是启动哪些服务。因为这些信息对于系统启动是至关重要的，所以Windows 2000/XP对这部分信息加以备份。当由于改变的配置而使得系统不能启动时，管理员可以选择原先的备份来重新启动系统。

5. HKEY_PERFORMANCE_DATA

注册表提供HKEY_PERFORMANCE_DATA根主键使得用户可以访问Windows 2000的系统性能评估数据，其中包括操作系统组件和服务器的性能数据。这个主键在注册表编辑器里是不可见的，只能通过Win32API的注册表函数来操纵它。

6. HKEY_CURRENT_CONFIG

HKEY_CURRENT_CONFIG主键只是一个指向当前硬件配置文件的链接，这个文件存储在“HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current”下。硬件配置文件使管理员能为系统的基本驱动程序设置参数。尽管配置文件可能在每次启动时有所改变，但应用程序总能通过这个主键的键值取得当前活动的配置文件。

2.6 Windows 2000/XP服务

几乎每一个操作系统都有一个在系统启动时自动运行服务进程的机制。在Windows 2000/XP里，这些进程被称之为服务（service）。这些服务进程类似于UNIX的守护进程，在客户/服务器应用程序中扮演服务器角色。一个Win32服务的例子就是Web服务器，由于它不论任何用户登录计算机都必须运行，而且在系统启动的时候就会开始运行，因此管理员不必记得甚至不用经过提醒就可以启动它。

Win32服务由三部组成：一个服务应用程序；一个服务控制程序（SCP）；服务控制管理器（SCM）。

2.6.1 服务应用程序

类似Web服务这样的服务应用程序，至少由一种Win32可执行程序组成。用户用SCP来启动、停止或者配置服务。虽然Windows 2000/XP内置的SCP提供集成好的功能来实现对一般的启动、

停止、暂停和继续的控制，但是一些服务应用还是包括了它们自带的SCP以便允许管理员来指定他们针对特殊服务的配置。

简单的服务应用程序类似于Win32可执行程序，其中有附加的代码接收来自SCM的命令，并且将服务应用程序的状态反馈给SCM。由于大多数的服务应用没有用户界面，因此它们被编译成控制台程序。

当安装了包含服务的应用程序后，安装程序必须向系统注册这个服务。安装程序调用Win32的CreateService函数实现注册。Asvapi32（高级应用程序接口动态链接库，Advanced API DLL）包括了所有的客户端SCM应用程序接口。

注册了一个服务之后，一条消息将会被发送到SCM中服务的所在地，然后SCM将为服务在“HKLM\SYSTEM\Currentcontrolset\Services”新建一个注册主键。每一个服务的独立的注册主键定义了可执行程序的包括服务、参数、配置内容的镜像的路径。

新建服务后，可以通过StartService函数启动一个安装或管理应用程序。因为一些基于服务的应用程序在启动时必须初始化，所以下述情况并不少见：安装程序将服务注册成为一个自动启动的服务，要求用户重启完成安装，由SCM在系统启动时启动服务。

当一个程序调用CreateService函数的时候，它必须指定描述服务特征的一系列参数。服务的特征包括：服务类型；服务可执行的程序镜像文件的地址；一个可选择显示名；一个可选择的帐号名和密码用来启动在一个特殊的帐号安全关联的服务；启动类型指明在系统启动时还是在SCP的指示下自动启动；错误处理代码则指明在服务启动时监测到有错误的情况下系统的对策，并且包括了假如服务为自启时当此服务与其他服务有关联启动的时候指定的可选信息。

SCM在服务注册表主键中以键值的形式保存了每种服务特征。如果服务需要保存服务的私有配置信息，通常会建立一个被称为参量的子键，在此子键中以键值的形式保存配置信息。此项服务就可以通过标准的注册表函数找到这些配置信息。

当SCM启动一个服务进程的时候，这个进程立刻调用StartServiceCtrlDispatcher函数，此函数接收一个服务入口列表或单个服务进程的单个入口。每个入口点通过与入口通信的服务名来鉴别。在建立了一个命名管道同SCM通信之后，此函数陷入循环等待来自管道的SCM的命令。SCM在每一次启动进程所属的服务时发送一个服务启动命令。而StartServiceCtrlDispatcher函数每收到一次服务启动命令就创建一个线程（叫做服务线程）来调用启动服务的入口和执行服务的循环命令。

StartServiceCtrlDispatcher不确定的等待来自SCM的命令，在所有进程的服务线程都停止并允许进程在其离开时清除资源后，才将控制权交还进程的主函数。

服务的入口点的第一个动作是调用RegisterServiceCtrlHandler函数。这个函数接收和保存了一张这个服务用以处理来自SCM的各种命令的函数列表。RegisterServiceCtrlHandler并不同SCM通信，但是它为StartServiceCtrlDispatcher函数在本地进程内存中保存了这张表。然后，服务入口继续初始化服务，包括分配内存、创建通信终端、从注册表中读取私有配置信息。在初始化服务的同时，入口点可能会阶段性地发送状态信息给SCM表明服务的启动的执行情况。在初始化结束后，服务线程通常陷入循环等待来自客户端的请求。以Web服务为例，它会初始化一个TCP监听socket，然后等待正确的HTTP连接请求。

服务进程的主线程由StartServiceCtrlDispatcher函数执行，它接收在服务进程中指示的SCM命令并且使用处理函数的列表定位和调用可信的服务函数来响应命令。SCM命令包括停止、暂停、恢复、查询和关闭等，当然也包括应用程序定义的命令。图2-15给出了一个服务进程的内在组成，描述了一个双线程进程组成的一个服务：主线程和服务线程。

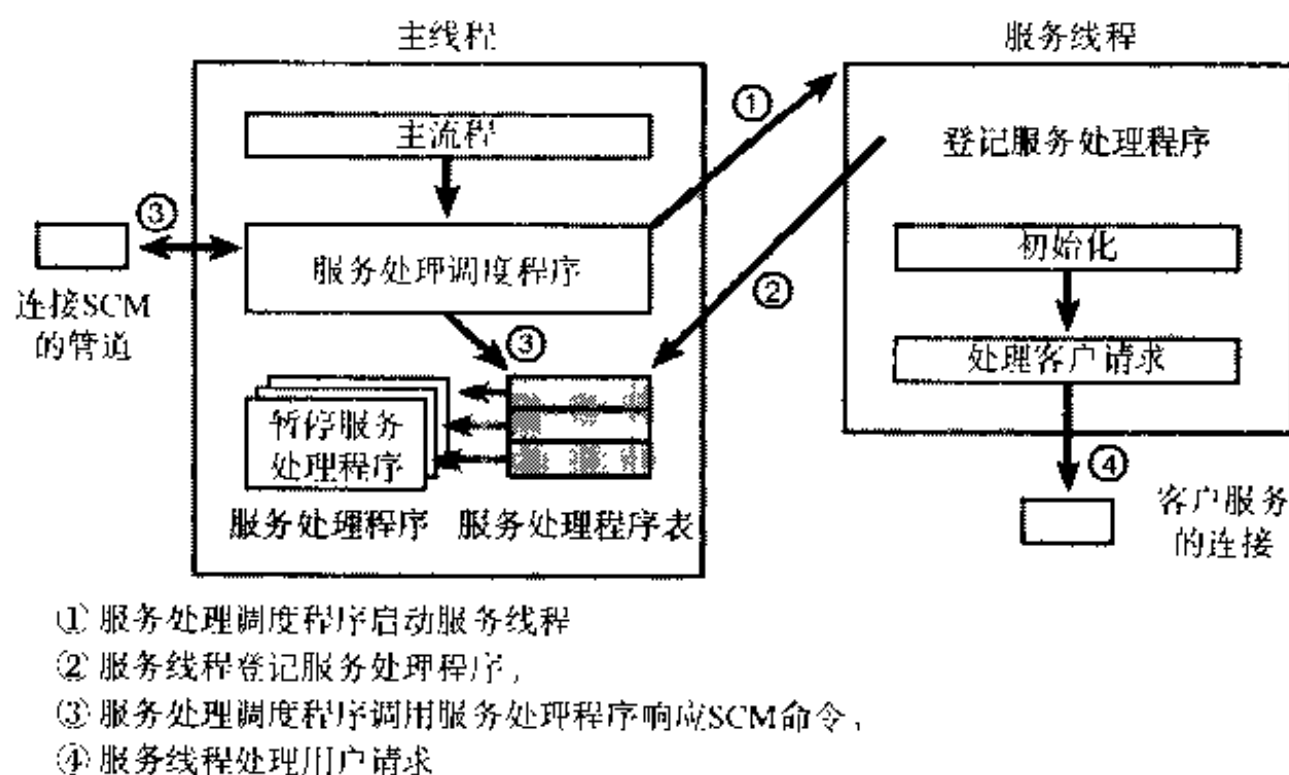


图2-15 服务的组成

2.6.2 服务帐号

对于服务开发者和系统管理员来说，服务的安全环境是一个非常重要的考虑因素。一般来说，除非服务的安装程序或管理员指定，服务运行于本地系统帐号的安全环境下。

1. 本地系统帐号

本地系统帐号与所有Windows 2000的用户模式操作系统组件运行的帐号一样，包括会话管理器、Win32子系统进程和Winlogon进程。从安全的远景来看，本地系统帐号非常强大——在本地系统上它比任何本地或域的帐号在本地安全权利都要大。这种帐号是本地管理员组的一个成员，具备授权所有实质性特权的权利。许多文件和注册表主键赋予本地系统帐号完全的存取访问权限。在本地系统帐号下运行的进程运行于缺省用户的状态（HKU\DEFAULT），因而不能访问保存在其他用户帐号里的配置信息。

当系统是Windows 2000/XP的一个域成员时，本地系统帐号就包含了服务进程运行于计算机的安全标志（Security Identifier, SID）。因此，通过计算机域帐号，运行于本地系统帐号的服务将和其他域树中的计算机中的服务一起被自动鉴别出来。除非机器帐号被明确地赋予了对资源的访问存取权限，否则进程可以访问允许无会话的网络资源。

2. 备用帐号运行服务

一些服务由于我们所描述的限制，必须有用户的帐号的安全证书才可以运行。可以配置一个服务运行于备用帐号，假定此服务由一个帐号和密码所创建或指定。服务必须在Windows 2000

服务MMC snap-in下运行。

2.6.3 交互式服务

另一个在本地系统帐号下运行服务的限制是它们不能在交互式的用户桌面上显示对话框或者窗口。这个限制并不是在本地系统帐号下运行的直接结果，而是服务控制器指派服务进程给Windows工作站（Windows station）的方法导致的结果。

Win32子系统用Windows工作站把每一个Win32进程联系起来，它包含了桌面，而桌面包含了视窗。在一个控制台上只有一个Windows工作站可见并且接收用户鼠标、键盘的输入信息。在服务环境终端，每一个Windows工作站的会话都是可见的，但是服务都是作为控制台会话的一部分运行。Win32把这种可见的Windows工作站称之为WinSta0，并且所有交互式的进程都对其进行访问。

除非有其他的命令，否则SCM把服务用一个不可见的名为Service-0x0-3e7\$的Windows工作站联系起来，这个Windows工作站是所有非交互式的服务所共享的。

这个Windows工作站名字中的数字——3e7——代表了LSASS指派给登录会话的会话登录标志，登录会话被SCM用来在本地系统帐号下运行非交互式的服务。

被配置成在用户帐号状态下运行的服务运行于有区别的不可见Windows工作站状态，并以Lsass指派给登录会话的标志命名。

无论是运行于用户帐号还是本地系统帐号下，没有运行于可见的Windows工作站下的服务都不能在控制台上接收输入或者显示窗口。事实上，如果一个服务在此Windows工作站下弹出了一个普通对话框，那么此服务将会表现出挂起状态，因为没有用户能见到这个对话框，所以当然阻止了用户用鼠标或键盘关闭它而使此服务继续执行下去。

虽然不常见，一些服务还是需要通过对话框或窗体同用户进行交互。一个含有这种需要的Windows 2000/XP内置的服务例子就是Windows 安装程序。为了配置一个拥有同用户交互权利的服务，在服务注册的主键类型参数中必须含有SERVICE_INTERACTIVE_PROCESS modifier。

当SCM启动一项标记为交互式的服务时，它在本地系统帐号安全环境下启动服务进程，并且把服务连接到Winsta0而不是非交互式的Windows工作站。这项连接允许服务在控制台上显示对话框和窗体的对用户的输入反馈。

2.6.4 服务控制器

SCM的可执行文件是“\Winnt\System32\Services.exe”，和许多服务进程一样，它以Win32 控制台程序模式运行，Winlogon进程在系统启动早期启动SCM。SvcCtrlMain在紧接着屏幕变为空白桌面的时刻运行，通常在Winlogon装载图形化身份鉴定并给出登录界面（GINA）之前运行。

SvcCtrlMain首先创建一个nonsignaled初始化的名为SvcCtrlEvent_A3752DX的同步事件，只有在完成准备接收来自SCP的命令的必要步骤后，SCM才设定此事件为signaled状态。SCP通过一个对话来确认SCM的函数是OpenSCManager。这个函数通过等待SvcCtrlEvent_A3752DX变成signaled来阻止SCP试图在SCM初始化完成之前接触SCM。

然后SvcCtrlMain开始工作并调用ScCreateServiceDB, 这个函数建立了SCM的内部服务数据库。ScCreateServiceDB存取“HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List”中的内容, 这是一个列出了定义好的服务组名称和命令的REG_MULTI_SZ值。服务的注册表主键值包括了可选的组键值, 表示服务或者设备驱动是否需要从其他的服务组角度考虑而控制它的启动顺序。举例来说, Windows 2000的网络堆栈是从下向上建立的, 所以, 在它们的启动顺序中, 网络服务必须指定组键值来把它放到网络设备的后面。SCM内建了一个组列表保存了从注册表键值读出的组顺序。组包括NDIS、TDI、Primary Disk、Keyboard Port和Keyboard Class。附加的和第三方的应用软件可以定义自己的组并把它们加入列表。微软公司的事件服务器就加了一个名为MS Transactions的组。

ScCreateServiceDB随后搜索“HKLM\SYSTEM\CurrentControlSet\Services”里的内容, 在服务的数据库里为每一个遇到的主键创建一个条目。一个数据库的条目包括所有的为一个服务定义好的服务关联参数和跟踪服务状态的域。由于SCM启动标记为自启的服务和设备驱动并且监测标记为引导启动和系统启动的驱动器启动错误, 因此SCM为服务和设备驱动加入了这些条目。在所有用户模式进程执行前, I/O管理器装载标记为引导启动和系统启动的驱动器, 因此任何有这些启动标记的驱动器将在SCM启动前被装载。

ScCreateServiceDB读取服务的组键值来确定此服务在组中的成员资格, 并且把这个键值同早先被建的组列表联系起来。这个函数还通过查询服务的DependOnGroup和DependOnService注册表键值在数据库中读取记录服务的组和从属信息。

在服务启动的时候, SCM有可能需要调用Lsass, 所以SCM等待Lsass在其初始化结束时通知LSA_RPC_SERVER_ACTIVE同步事件, Winlogon也启动Lsass进程, 所以SCM同Lsass的初始化是同步的, 而且它们两个的初始化结束顺序是不确定的。SvcCtrlMain调用ScGetBootAndSystemDriverState遍历服务数据库来查询引导启动和系统启动的设备驱动器条目。ScGetBootAndSystemDriverState通过查询驱动器在对象管理器中的名字域目录\Driver中的名字来确定驱动器是否成功的启动。当一个设备驱动器被成功装载, I/O管理器在它的目录下把驱动器对象插入到名字域, 所以如果它的名字没有给出, 那么它就不能被装载。如果一个驱动器没有被装载, SCM在PnP_DeviceList函数返回的驱动器列表中查询它的名字。PnP_DeviceList支持被包括在当前系统通用硬件配置中的驱动器。SvcCtrlMain记录了没有启动的驱动器的名字, 并作为ScFailedDrivers列表中当前配置文件的一部分。

在启动自启服务之前, SCM做了一些额外的事。它创建了叫做管道的远程过程调用Pipe\Ntsvcs, 并且开始一个线程来监听来自SCP的消息, 然后通知它的初始化结束事件SvcCtrlEvent_A3752DX。SCM通过由RegisterServiceProcess注册一个控制台应用程序关闭事件处理并向Win32子系统进程注册来为系统关闭做准备。

1. 服务启动

SvcCtrlMain调用SCM的ScAutoStartServices来启动所有有自启动初始值的服务, 也装入自启动的设备驱动程序。它按正确的顺序启动服务进程, 算法是阶段性执行的, 每一个阶段由此符合每一组, 并且由组次序定义好的顺序执行, 它保存在“HKLM\SYSTEM\CurrentControlSet\

Control\ServiceGroupOrder\List”注册表主键值中。因此，除考虑对属于其他组的服务有妨碍的情况下对服务启动的调整之外，把一个服务添加到一组是没有什么影响的。

当一个阶段开始时，ScAutoStartServices为了启动而把所有属于此阶段的组的服务条目做标记。然后它循环所有标记过的服务，检测是否每一个服务都能启动。检测内容包括确定此服务的启动是否依赖于其他组的服务，这由服务的注册主键值中的DependOnGroup是否存在决定。如果依赖关系存在，服务所依赖的组必须已经完成初始化，并且那组中至少有一个服务已经成功地启动了。如果在组启动顺序中一个服务所依赖的组比服务所在的组靠后，SCM标记一个“circular dependency”错误给服务。如果ScAutoStartService考虑一个Win32服务而不是设备驱动器，它下一步会检测此服务依赖于一项还是多项服务，这些服务是否都已经启动。服务依赖关系在服务的注册主键值的DependOnService键值中被指定。如果一个服务所依赖的其他服务属于在其后启动的组，SCM同样产生一个“circular dependency”错误并且不启动这个服务。如果服务依赖于本组的任何未启动服务，则此服务的执行将会被跳过。

当服务的依赖关系通过检测后，在启动服务之前，ScAutoStartServices为此服务是否为当前通用引导配置的一部分做最后的检测。若系统以安全模式启动，SCM确保服务在正确的安全引导注册主键值中通过名字或组被鉴别出来。在“HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot”中，有两种安全引导的键值：最小配置和网络支持配置，启用哪种配置则由用户所选的安全模式类别来决定。如果用户在专用启动菜单中选取普通安全模式或命令行安全模式，SCM将启用最小配置；如果用户选取网络安全模式，SCM将启用网络支持配置。在安全启动的键值中Option的值不仅指明系统以安全模式启动，还指明用户选择的安全类别。

一旦SCM决定启动一项服务，它调用ScStartService，ScStartService对服务和设备驱动器的启动采取不同的步骤。当ScStartService启动一个Win32服务时，它首先读取服务的注册主键值中的ImagePath来确定运行服务进程的文件名。然后检查服务类型值，其值是否为SERVICE_WIN32_SHARE_PROCESS (0x20)，SCM确保插入的服务进程与已启动的服务用相同的帐号登录。服务的注册键值中的对象名保存了服务运行的帐号信息。没有对象名或对象名为本地系统的服务运行于本地系统帐号环境下。

在SCM内部的image database数据库中，SCM通过检查服务的ImagePath是否有一个条目来核实服务进程是否在另外的帐号下运行。如果image database不存在ImagePath值的条目，SCM将会创建一个，并且保存服务所用的登录帐号和从服务的ImagePath得来的数据。SCM要求服务含有一个ImagePath值，如果没有，SCM报告一个它不能找到服务路径和不能启动服务的错误。如果SCM定位了一个存在的image database条目和匹配的ImagePath数据，SCM确保服务启动的用户帐号信息同保存在数据库里的信息一致——一个进程仅能以一个帐号登录，所以在同一个进程中，一个服务指定的帐号同其他已启动的服务在此进程中指定的帐号不同时，SCM将会报告错误。

如果服务指定了配置，SCM调用ScLogonAndStartImage登录服务并启动服务进程。SCM调用Lsass函数来登录不在系统帐号下运行的服务。当SCM调用LsaLogonUser，它指派服务的登录类型，Lsass在注册表子键Secrets形如_SC_<service name>的条目下寻找密码。当SCP配置服务的登录信息时，SCM指导Lsass隐式地保存登录密码。当成功登录时，LsaLogonUser返回一个调

用者访问的句柄。Windows 2000使用访问标记来代表用户的安全权限，SCM稍后把实现服务的进程同访问标记联系起来。

成功登录后，如果没有装载帐号信息，则SCM通过调用UserEnv DLL的LoadUserProfile 函数来装载帐号的信息。“HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList\<user profile key>\ProfileImagePath” 值包括了LoadUserProfile函数装载到注册表的registry hive磁盘地址，为在hive中的HKEY_CURRENT_USER创建信息。

交互式的服务必须打开WinSta0，但是在ScLogonAndStartImage允许交互式服务访问WinSta0之前，它检查是否已设定“HKLM\SYSTEM\CurrentControlSet\Control\Windows\NoInteractiveServices”值。管理员设定这个值来阻止交互式服务在控制台上显示窗口。这个选项满足了无人服务器环境的需要，此环境下交互式服务中并没有用户可以反馈。

下一步，如果服务进程没有被启动，ScLogonAndStartImage会继续启动服务进程。SCM用CreateProcessAsUser Win32函数以挂起状态启动进程，然后创建一个命名管道来同服务进程通信，并把管道命名为“\Pipe\Net\NetControlPipeX”，每一次SCM创建一个新管道都对X加一。SCM通过ResumeThread函数来恢复服务进程和等待服务连接到SCM管道。如果“HKLM\SYSTEM\CurrentControlSet\Control\ServicesPipeTimeout”键值存在，它决定了SCM等待服务调用StartServiceCtrlDispatcher的超时时间。如果ServicesPipeTimeout键值不存在，SCM使用30秒作为缺省时间，SCM在所有的服务通信中使用相同的超时时限。

当服务通过管道同SCM连接上时，SCM发送一个启动命令给服务。在超时时限内，如果服务没有对启动命令做出明确响应，SCM将放弃并开始启动下一个服务。在超时时限内，服务对启动命令没有响应时，SCM并不像服务没有调用StartServiceCtrlDispatcher那样终止进程，而是在系统事件日志中记录一个错误，说明此服务没有在时限内成功启动。

如果SCM 调用ScStartService启动的服务的注册表主键值类型为SERVICE_KERNEL_DRIVER或服务SERVICE_FILE_SYSTEM_DRIVER，此服务为设备驱动器，所以ScStartService调用ScLoadDeviceDriver来装载驱动器。ScLoadDeviceDriver为SCM进程启用安全特权，然后调用内核服务NtLoadDriver，在ImagePath中传递驱动器注册主键表值的数据。与服务不同，驱动器不需要指定一个ImagePath值，如果此值不存在，SCM通过添加驱动器名到“\Winnt\System32\Drivers\”字符串来建立一个image path。

ScAutoStartServices持续循环组内的服务直到所有的服务不是启动就是发生了依赖关系错误为止。这种循环是SCM依照服务的DependOnService依赖关系自动排列服务的次序的方法。SCM将会先循环其他服务所依赖的服务，跳过依赖于其他服务的服务直到子序列开始循环。注意SCM忽略Win32服务的标志符，在“HKLM\SYSTEM\CurrentControlSet\Services”键值里可以看到这个标志符；而I/O管理器则以此标识符来排列有关引导和系统启动的设备驱动器的启动。

一旦SCM结束了在ServiceGroupOrder\List中所有列出的服务组的所有阶段，接着它执行属于列表中没有列出的组的服务，最后一步执行不属于任何组的服务。

2. 启动错误

如果驱动器或服务对SCM启动命令回应一个错误，在服务注册键值中的ErrorControl决定了

SCM如何处理。如果其值为SERVICE_ERROR_IGNORE (0) 或没有被指定, SCM只是简单地忽略此错误并继续服务的启动。如果其值为SERVICE_ERROR_NORMAL (1), SCM写入一个事件到系统事件日志, 描述为“<服务名>服务由于下面的原因启动失败:”, 记录在事件日志记录中, SCM包括了服务作为启动失败原因反馈给SCM的文本Win32错误类型代号。

如果含有SERVICE_ERROR_SEVERE (2) 或SERVICE_ERROR_CRITICAL (3) ErrorCtrol值的服务报告一个启动错误, SCM写入一个记录到事件日志并且调用内部函数ScRevertToLastKnownGood。此函数切换系统注册配置到一种被称为last known good的版本, 就是系统上一次成功启动所用的版本。然后SCM用NtShutdownSystem系统服务重启系统, 此服务在可执行程序中已经实现。如果系统已经使用last known good配置启动, 那么系统只是简单地重启。

3. 接受最近一次成功引导的配置

除了启动服务, 系统还管理SCM决定系统注册配置HKLM\SYSTEM\CurrentControlSet何时被存为last known good control控制设置。CurrentControlSet把服务主键作为一个子键包含, 所以它包括了SCM数据库中的注册表, 还包括了保存许多核心态和用户态子系统的配置信息的控制主键。缺省情况下, 一次成功的启动包括所有自启服务的成功启动和一个用户的登录。如果系统在启动时因为设备驱动器崩溃或带有SERVICE_ERROR_SEVERE 或 SERVICE_ERROR_CRITICAL ErrorControl值的自启服务报告一个启动错误而导致异常终止, 则此次为一次失败的启动。

SCM显然能确信它成功地完成了一次自启服务的启动。但是对于Winlogon (\Winnt\System32\Winlogon.exe), 则必须把一次成功的登录通知它。当用户登录时, Winlogon调用NotifyBootConfigStatus函数, 此函数发送一个信息给SCM。在所有自启服务成功启动和收到来自NotifyBootConfigStatus函数的登录信息 (任何最后到来的一个) 后, SCM调用NtInitializeRegistry保存当前的启动注册配置信息。

第三方软件开发者可以用他们自己的定义来取代Winlogon的定义。举例来说, 运行Microsoft SQL Server的系统可能直到SQL服务器能接受和处理事务时才确定启动为成功。开发者通过写一个启动验证程序并且把程序安装到保存在注册表主键值“HKLM\SYSTEM\CurrentControlSet\Control\BootVerificationProgram”中的磁盘地址来加入他们对系统成功启动的定义。另外, 启动验证程序的安装必须通过设定“HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\ReportBootGk”为0来禁用Winlogon对NotifyBootConfigStatus的调用。当启动验证程序被安装时, SCM在启动完所有自启服务后才运行它, 并且在保存last known good控制配置信息前等待程序调用NotifyBootConfigStatus。

Windows 2000保留了多份CurrentControlSet的拷贝, 而CurrentControlSet实际上只是一个指向一份拷贝的象征性的链接。控制配置在“HKLM\SYSTEM\ControlSetnnn”表单里都有名字, 其中nnn是001、002这类的数字。HKLM\SYSTEM\Select键值包括了鉴别每一个控制配置的值。比方说, 如果CurrentControlSet指向ControlSet001, 那么当前选定的值为1。选定的LastKnownGood值包括了上一次成功启动时已知的正确的控制配置。在你的系统上另一个值可能是失败的, 指向上一次没有成功启动并且使用last known good 控制设置启动的控制设置。

NtInitializeRegistry从last known good 控制设置中读取内容并且把它同CurrentControlSet key

树同步起来。如果这是系统的第一次成功启动, last known good并不会存在, 于是系统会为它创建一个新的控制设置, 如果它存在, 系统只是简单地把它同CurrentControlSet不同的部分做更新。

Last known good在CurrentControlSet有变更的时候很有帮助, 就像在HKLM\SYSTEM\Control里改变了系统的performance-tuning值或者增加了一个服务或设备驱动器而导致后来的系统启动失败的情况。用户可以在系统引导进程的早期按下F8来引出一个菜单, 使他们可以使用last known good控制设置, 把系统配置重置为上一次成功启动时的正确配置。

4. 服务失败

一个服务可以在它的注册表主键值里含有可选的失败处理动作和失败处理命令, 在系统启动时, SCM记录这个注册表主键值。SCM向系统注册, 所以系统在一个服务进程存在时通知SCM。

服务可以为SCM配置的失败处理动作包括重启这个服务、运行一个程序或者重启计算机。此外, 服务可以指定服务进程第一次失败、第二次失败和接下来的失败时的失败处理动作, 并且如果服务被要求重启, 还可以指定SCM在服务重启前等待的延时。IIS服务管理器的失败处理动作将导致SCM运行IISReset应用程序, 清除所有工作并且重启服务。你可以很方便地在Services MMC snap-in里的服务属性对话框里的Recovery标签下管理上述的恢复操作。

5. 服务关闭

当被Winlogon调用时, Win32ExitWindowsEx函数发送一条消息给Win32子系统进程Csrss, 调用Csrss的关闭例程。Csrss遍历所有活跃进程并且通知它们系统正在关闭。在通报下一个进程前, Csrss等待除了SCM以外的每一个系统进程退出, 等待的秒数在“HKU\DEFAULT\Control Panel\Desktop\WaitToKillAppTimeout”里指定(缺省值为20秒)。当Csrss遇到SCM进程时, 它也通报SCM系统正在关闭, 但是等待专为SCM指定的超时。在系统初始化时, SCM通过RegisterServicesProcess函数向Csrss注册它的进程ID, Csrss便通过使用SCM的进程ID(Csrss保存这个ID)辨认SCM。SCM的超时之所以与其他进程不同是因为Csrss知道SCM还要与需要在关闭时清除数据的服务进行通信, 所以管理员可能仅需要调整SCM的超时。SCM的超时值在“HKLM\SYSTEM\CurrentControlSet\Control\WaitToKillServiceTimeout”中, 并且缺省状态设为20秒。

SCM的关闭处理程序有责任发送关闭通知给所有那些在SCM初始化时申请需要关闭遍知的服务。SCM的ScShutdownAllServices遍历SCM服务数据库寻找那些请求有关闭遍知的服务并发送关闭通知给每一个这样的服务, 并为每一个这样的服务记录服务的等待延时值, 这个值在服务向SCM注册时就指定了。SCM明确它所收到的最大的等待延时。发送关闭通知后, SCM等待它所通知的服务退出或者一直等待到超过最大的等待延时为止。

如果服务在超过等待延时后仍没有退出, SCM就测定一个或多个它正在等待退出的服务是否已经发送了一个消息给SCM, 告诉SCM此服务在它的关闭进程上取得进展。如果至少一个服务有进展, SCM就在等待延时的范围内再等待一次。SCM持续执行它的等待循环直到所有的服务都已经退出或者它所等待的服务都没有在等待延期内发送给它取得进展的信息为止。

当SCM忙于通知服务关闭并且等待它们退出的时候, Csrss等待SCM退出。如果Csrss等待超时而SCM没有退出, Csrss简单转移, 继续它的关闭进程。因而, 在系统关闭时, 那些在规定时间内没有成功关闭的服务只是简单地同SCM一起执行。不幸的是, 管理员无法知道他们是否应该

为那些在系统关闭前无法完全关闭的服务增加WaitToKillServiceTimeout值。

6. 共享服务进程

在Windows 2000/XP自带的内部服务中，有些服务运行于它们自己的进程空间，而有些服务运行于与其他服务共享的进程空间。举例来说，SCM进程上运行了事件日志服务、文件服务器服务（LanmanServer）和LAN Manager name resolution服务。

有一种称为Service Host的进程包含了多个服务。多个SvcHost的实例能被运行于不同的进程空间。运行在SvcHost进程的服务包括Telephony（TapiSrv）、Remote Procedure Call（RpcSs）和Remote Access Connection Manager（RasMan）。Windows 2000以DLLs的形式执行在SvcHost里运行的服务并且在服务的注册表主键值中包括了一个ImagePath的定义。服务的注册表主键值也必须在子键值中包括名为ServiceDll的键值，指向服务的DLL文件。

所有共享一个SvcHost进程的服务指定一个同样的变量，所以它们在SCM的数据库中只有一个条目。在服务启动时，如果SCM遇到第一个含有特殊参数的SvcHost ImagePath的服务，它创建一个新的数据库条目并且使用这个参数执行SvcHost 进程。新的SvcHost进程采用这个参数并且在HKLM\ SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost下寻找与它同名的值，SvcHost读取这个值的内容，将它解释为一串服务名的列表，并且在SvcHost向SCM注册时通知SCM它正在提供这些Service Host服务。

当SCM在服务启动时遇到SvcHost服务，并且这个服务在启动时带有与image数据库匹配条目的ImagePath时，它并不启动另一个进程而只是为服务发送一个启动命令给带有那个Imagepath值的已经启动的SvcHost。于是，这个SvcHost进程在服务注册键值中读出ServiceDll参数并且装载此DLL到进程来启动服务。

7. 服务控制程序

服务控制程序都是标准Win32应用程序，使用CreateService、OpenService、StartService、ControlService、QueryServiceStatus和DeleteService SCM 函数。为了使用这些SCM函数，SCP必须首先通过调用OpenSCMManager函数打开一个同SCM通信的通道。在调用的时候，SCP必须指定它需要执行的动作类型。比方说，如果SCP只是简单地列举SCM数据库里的当前服务，它在调用OpenSCMManager的时候就请求列举服务。在初始化的时候，SCM创建了一个内部的对象代表SCM数据库，并且用Windows 2000/XP的安全函数通过安全描述符来保护这个对象，安全描述符里指定了帐号的安全权限。

SCM执行于服务自身的安全环境，当SCP通过CreateService函数创建一个服务时，它指定一个安全描述符，此描述符是同SCM服务数据库中服务的条目有内在关联的。SCM保存在服务注册主键值中的安全描述符为安全值，在初始化时期，当它浏览服务注册表主键值的时候读取这个值，使得重新启动期间安全设置一直持读。同样的，SCP必须在调用OpenSCMManager时指定它将对SCM数据库访问的访问类型，SCP必须在调用OpenService中告诉SCM它将对服务进行的访问类型。SCP请求的访问类型包括查询服务的状态，配置、停止和启动一个服务。

大家最熟悉的SCP可能是Windows 2000里的MMC snap-in服务，它在“\Winnt\System32\Filemgr.dll”中。SCP有时在SCM执行之上把服务策略分层。一个很好的例子就是在手动启动一个

服务时MMC snap-in执行的超时服务。Snap-in给出一个进度条来显示服务启动的进度。服务通过设定它们的配置状态来间接地同SCP交互，它们的配置状态反映了服务对SCM类似于启动命令的命令的响应进展。SCP通过QueryServiceStatus函数查询配置状态。它们可以告诉服务：在服务处于挂起状态时动态地更新这些状态，并且SCM可以采取适当的动作通知用户服务当前的工作情况。

2.7 Windows 2000/XP的管理机制

Windows 2000/XP通常利用事件管理器报告错误信息和诊断信息。而事件查看工具则能使管理员详细掌握本地机和远程机器上的事件动态。与之相似的是，性能评估机制使得应用程序和操作系统组件能将性能表现的统计数据传送给性能监视器（Performance Monitor）。

Windows NT事件监视器和性能监视器是有一定局限性的。举例来说，程序接口千差万别，这种差异无疑增加了应用程序运用事件和性能管理器收集数据的复杂程度。而性能评估机制体现不出它们之间的细微差别，尤其是通过网络的时候，因为这些性能统计数据往往经过系统的修改，所以不能直接反映你所关心的对象的情况。Windows NT 4监视系统中最大的弱点就是它们几乎不具备可扩展性，而且日志和性能数据的收集程序也没有给用户交互中所必需的编程接口。应用程序必须以预先定义好的格式提供数据，而程序没有办法收到与性能评测相关的事件的提示消息，而且事件管理器所提供的事件的消息也无法让程序知道这些事件的确切类型和来源。最后，这些性能数据评估程序实际上无法通过事件管理器或是应用程序接口与事件或性能数据的提供程序进行通信。

为了消除这些局限，同时也为其他的数据源提供管理和分析的工具，Windows 2000/XP引入了一种新的机制：Windows Management Instrumentation（WMI）。WMI是基于网络的企业管理（Web-Based Enterprise Management，WBEM）标准的实现，这个标准是由分布式管理工作小组（Distributed Management Task Force，DMTF）所制订的。WBEM标准中包括了事务数据收集和管理的设计，这种设计具有很强的伸缩性和扩展性，这两种特性正是管理可能由多种不同组件组成的本地机和远程机器所必需的。

2.7.1 WMI的体系结构

· WMI包含4个主要的组件（如图2-16所示）：管理程序、WMI基础设施、数据生产者和被管理的对象集合。管理程序处理和显示应用程序从被管理的对象获取的数据。一个简单的例子是性能评测工具使用WMI标准而不是API来收集性能相关的统计数据，而更复杂的例子是，一个基于WMI标准的企业级的管理工具可以使管理员方便地管理每台软件与硬件的配置。

开发者往往将管理应用程序的目标定位于从指定的对象获取数据。一个对象可能代表一个组件，如一个网络适配器，也可能代表一系列的组件，如一台计算机。数据生产者需要确定和输出管理应用程序所关心的对象在系统中代表什么。比如，一个网络适配器的厂商可能会将一些适配器的属性加入WMI所支持的适配器中，这样应用程序可以通过WMI根据自己的需要查询和设置适配器的状态和行为。在一些应用（如设备驱动程序）中，微软公司提供了一个带有编程接口的数据生产者，使开发者用最少的代码设计出拥有自定义的管理对象的数据生产者。

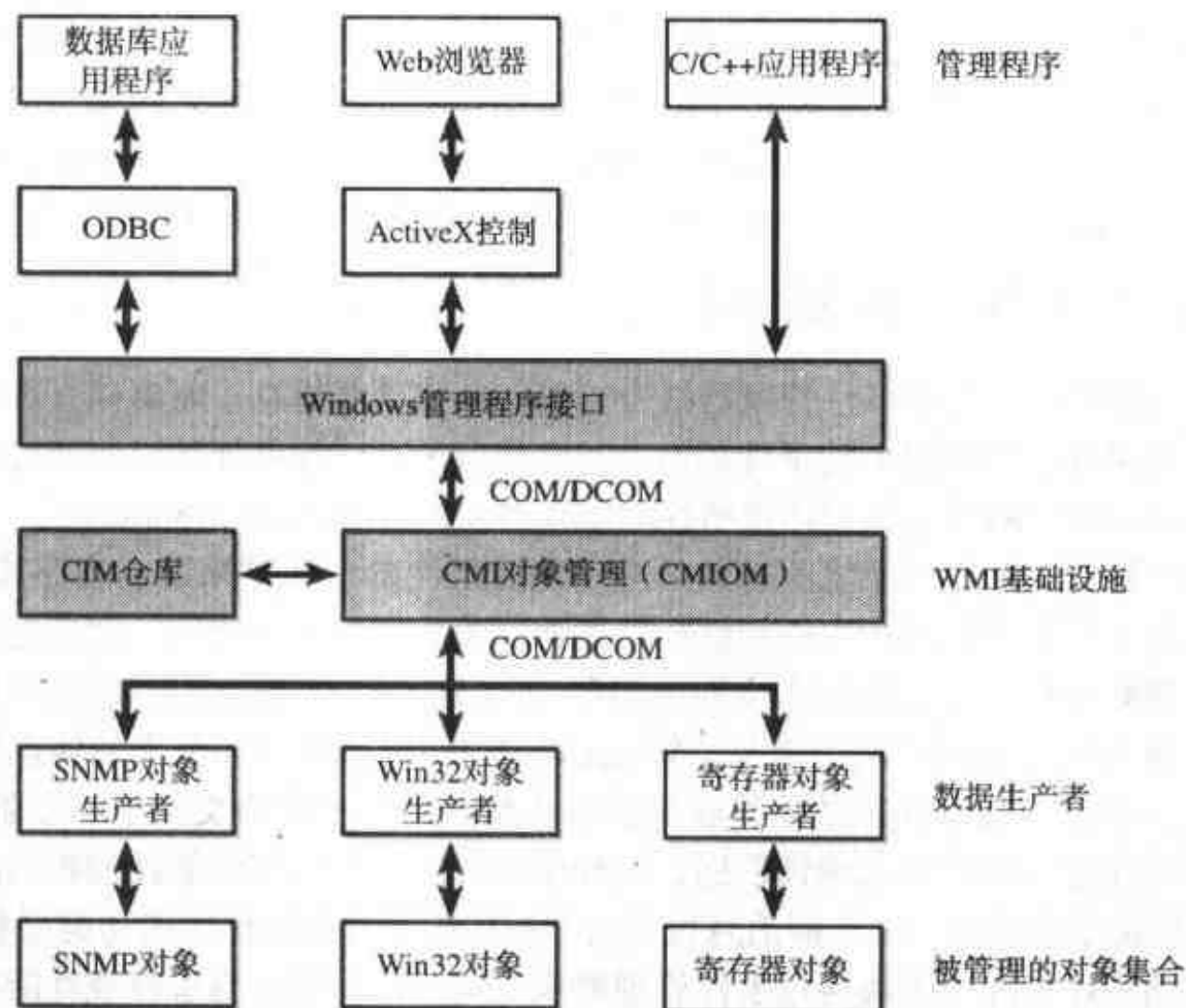


图2-16 WMI的体系结构

WMI基础设施是通用信息模型对象管理器（CIMOM）的核心，它是联系管理应用程序和数据生产者的桥梁。WMI将数据以一个名为CIMOM Object Repository的数据库的形式存储在磁盘上。作为这个基础设施的一部分，WMI同时也支持一些API，通过它们，管理应用程序就可以访问对象以及数据生产者所提供的数据和类的定义。

Win32程序使用底层的WMI-COM接口直接与WMI通信。其他接口都是在这层接口之上的，包括一个为Microsoft Access数据库应用程序设计的开放数据库连接（ODBC）适配器。数据库程序开发者用WMI ODBC适配器将数据源嵌入开发着的数据库中，这样就可以通过对基于WMI的数据的查询生成数据报表。WMI ActiveX控件支持另一个层次的编程接口，网络程序开发者可以用这些控件构造出访问WMI数据的交互界面。微软公司还提供还有一种WMI脚本API，用于那些基于脚本的应用程序和Microsoft Visual Basic程序；实际上微软公司所有的编程语言都支持这种脚本API。

同样在与数据生产者交互的过程中，WMI COM接口组成了WMI最基本的API。但不同的是，管理应用程序是COM的客户端，而数据生产者则是COM或DCOM的服务器端（也就是说，WMI中的COM对象是由数据生产者实现的）。WMI数据生产者的可能表现形式有：WMI管理进程加载的动态链接库、独立的Win32应用程序以及独立的Win32服务进程。微软公司提供了一套内建的数据生产者，它们提供了很常用的数据源，如性能API、注册表、事件管理器、活动目录、SNMP和基于WDM（Windows Driver Model）的设备驱动程序。同时，WMI的开发包使得我们可以开发第三方的WMI数据生产者。

2.7.2 数据生产者

WBEM的核心是基于DMTF设计的CIM。CIM规定了管理系统如何表示从计算机传递到应用程序或设备的信息。它还从系统级的层次上确定了管理系统的每一部分的描述，大至整个系统，小到一个应用程序或一个设备。这样，开发者就可以用CIM来定义自己所写的管理程序中所需的组件，通常用管理对象格式（Managed Object Format, MOF）语言来描述。

除了为对象定义类之外，数据生产者还必须为WMI提供这些对象的编程接口。WMI根据数据生产者提供接口的不同将它们分成若干类。注意，一个数据生产者可能同时具有多个特征，因此它可以同时是类和事件的生产者。为了说明这一点，看一个同时具有若干个特征的数据生产者——事件日志数据生产者，它定义了多个对象，包括一个事件日志计算机、一个日志记录和一个日志文件。因为它在定义这些对象时用到了类定义，所以它是一个类类型的数据生产者，必须将这些类的定义传给WMI。同时它又是一个实例类型的数据生产者，因为它为自己所属的好几个类定义了多个实例，其中一个类是事件日志文件类，这个数据生产者用这个类为每个系统日志（系统事件日志、应用程序事件日志和安全事件日志）都创建了一个实例。

事件日志数据生产者定义了实例数据，这样管理应用程序就可以列举它们。为了让应用程序用WMI存储和备份这些事件日志文件，数据生产者对日志文件对象提供了实现这些功能的方法。这样做使得这个数据生产者又是一个方法类型的生产者。最后，可以要求在每一个新的记录写入日志文件时，数据生产者都要给应用程序发送消息，这样这个数据生产者实际又有事件类型生产者的属性，因为它用到了WMI的事件响应机制。

2.7.3 通用信息模型和管理对象格式语言

CIM紧随着面向对象语言（如C++、Java）的步伐，这些语言以类的表示形式设计模板。类的运用使开发人员得以使用强大的继承和合成建模技术。子类可以继承父类的特性，并且可以增加它们自己的特性和重载它们从父类继承来的特性。类还可以组合，开发者可以创建一个包含其他类的类。

DMTF提供了多样的类作为WBEM标准的一部分。这些类是CIM的基本语言，代表了适用于所有管理领域的对象，是CIM核心模块的一部分。一个核心类的例子是CIM_ManagedSystemElement，这个类包含了很少的基本属性，基本属性标志了物理部件（如硬件设备），还标志了逻辑部件（如进程、文件）。这些属性包含了标题、描述、安装日期和状态。因而，CIM_LogicalElement和CIM_PhysicalElement类继承了CIM_ManagedSystemElement的属性，这两个类也是CIM核心模块的一部分。WBEM标准称这些类为抽象类，因为它们只为其他类继承而存在（也就是说，没有一个抽象类的对象实例存在）。我们由此可以把抽象类看作是定义了其他类使用的属性的模板。

第二种类代表了那些对于管理领域特殊但又并非特殊实现的对象。这些类组成了通用模块并被认为是核心模块的扩展。一个通用模块类的例子是CIM_FileSystem类，它继承了类CIM_LogicalElement的属性。因为实际上每一个操作系统，包括Windows 2000/XP、Linux和其他版本

的UNIX，都依靠文件系统存储，所以CIM_FileSystem类是通用模块中一个合适的要素。

最后一种类型给普通模型附加了一些面向具体技术的成分。Windows 2000定义了一大组这种类型来代表Win32环境特殊的对象。由于所有的操作系统在文件中保存数据，因此CIM通用模块包括了CIM_LogicalFile类。CIM_DataFile类继承CIM_LogicalFile类，而且Win32增加了Win32_PageFile和Win32_ShortcutFile文件类给这些Win32文件类型。

事件日志供应者使用了扩展继承属性。时间日志文件是一个数据文件，含有增加的特殊事件日志属性，比如一个日志文件名（LogfileName）和一个文件所包含记录的统计数字（NumberOfRecords）。类浏览器显示的类树揭示了Win32_NTEventlogFile是基于几个继承层次的，其中，CIM_DataFile从CIM_LogicalFile中派生得来，而CIM_LogicalFile从CIM_LogicalElement派生得来，CIM_LogicalElement又是从CIM_ManagedSystemElement派生得来的。

在MOF中构造一个类后，WMI开发者可以通过几种方法提供WMI对类定义的支持。WDM开发者把MOF文件编译成一个二进制MOF文件（BMF）——一个比MOF文件更简洁的表示法——并且把这个BMF文件交给WDM基础设施。另一种方法是编译MOF文件并且使用WMI COM APIs把类定义送给WMI基础设施。最后，生产者可以使用MOF编译工具（Mofcomp.exe）来直接交给WMI基础设施一个类编译好的代表。

2.7.4 WMI名字空间

类定义了对对象的属性，而在系统中对象是类的一个实例。WMI使用一个名字空间来组织对象，名字空间由几个子名字空间组成，WMI层次地排列这些子空间。一个管理应用程序在名字空间里访问类之前必须先连接上这个名字空间。

WMI命名名字空间的根目录为root。所有WMI有四个在root下预先定义好的名字空间：CIMV2、Default、Security和WMI。这些名字空间中的一些名字空间包括了其他的名字空间。比如，CIMV2包括了Applications和ms_409 namespaces作为它的子名字空间。生产者有时也定义它们自己的名字空间。

与文件系统的名字空间不同，文件系统是由目录和文件的分级结构构成，而WMI名字空间只有一个层次。WMI使用对象的属性而不是像文件系统那样使用名字，它定义这些属性鉴别对象。管理应用程序用关键字名指定类名以定位一个名字空间中的特殊对象。因此，每一个类的实例都必须通过关键字的值唯一确定。比如，事件日志生产者使用Win32_NTLogEvent类来描述事件日志中的记录。这个类有两个关键字：Logfile，一个字符串；RecordNumber，一个无符号整数。向WMI查询事件日志记录实例的管理程序从标志此记录的那对关键字得到它们。应用程序使用下面示例中对象路径名的语法来访问记录。名字开始的部分（\PICKLES）标志了对象所在的计算机，第二部分（\CIMV2）是对象所处的名字空间。类名紧跟在冒号后面，关键字名和它相关的值跟在其后，关键字值用逗号隔开。

2.7.5 类联合

许多对象类别都通过一些途径同其他对象类别相关联。WMI使供应者构造一个类联合来代表

两个类之间的逻辑连接。类联合把一个类同另一个类关联起来，所以它只有两个属性：一个类名和Ref modifier。给定一个对象，管理应用程序可以查询被关联的对象。通过这种途径，生产者定义了一个对象层次。

典型的Win32系统组件把它们的对象放在CIMV2名字空间里。对象浏览器首先定位Win32_ComputerSystem对象实例DSOLOMON，它代表了计算机，然后对象浏览器获得与此对象关联的其他对象并在DSOLOMON下展示出来。对象浏览器用户界面用一个双向箭头文件夹图标来展示关联对象，被关联的对象在这个目录下展示。

2.7.6 WMI对象浏览器

在对象浏览器里，你可以发现事件日志生产者关联类Win32_NTLogEventComputer在DSOLOMON的下面并且有众多的Win32_NTLogEvent实例存在。在对象浏览器里选择一个Win32_NTLogEvent类的实例将会在右边的面板的属性标签中展示类的属性。微软公司计划使用对象浏览器来帮助WMI开发者检查他们的对象，但是一个管理应用程序可以完成同样的操作并且可以以更易理解的方式展示或收集类的属性信息。

2.7.7 WMI执行

WMI基础设施主要在“\Winnt\System32\Wbem\Winmgmt.exe”文件中实现。这个文件以Win32服务方式运行，在一个管理应用程序或WMI生产者试图第一次访问WMI APIs时启动它。大多数WMI组件缺省驻留于“\Winnt\System32 and \Winnt\System32\Wbem”，包括Win32 MOF文件、内建的生产者动态链接库和管理应用程序WMI动态链接库。

“\Winnt\System32\Wbem”下的目录保存库、记录文件和第三方MOF文件。WMI执行库——称为CIMOM库——“\Winnt\System32\Wbem\Repository\Cim.rep”文件。Winmgmt认可众多同库关联的注册表设定（包括不同的内在性能参数，比方说CIMOM备份的地址和时间间隔），库的“HKLM\SOFTWARE\Microsoft\WBEM\CIMOM”注册表主键值保存了这些注册设定。

设备驱动器用特殊的接口来提供数据和接受从WMI来的命令——调用WMI系统控制命令。因为这些接口是跨平台的，所以它们归属于“\root\WMI”名字空间下。

2.7.8 WMI安全

WMI在名字空间层上执行安全措施。如果一个管理应用程序成功地连接上了名字空间，应用程序可以查看和访问所有处于那个名字空间的对象的属性。管理员可以使用WMI控制应用程序来控制何种用户可以访问一个名字空间。

习题

- 2.1 操作系统作为一个软件，具有什么样的特点？你能否举出相应的例子说明这些特点？
- 2.2 在设计操作系统的时候要考虑哪些因素的影响，为什么？
- 2.3 一个优秀的操作系统设计应该具备什么样的特点，如何理解这些设计目标？你认为

Windows 2000/XP是否具备这些特点，请举例说明。

- 2.4 操作系统设计的过程包括哪些阶段，在每个阶段都要考虑什么问题。
- 2.5 有哪些常见的操作系统的体系结构，它们各自都有什么特点？Windows 2000/XP采用了什么样的体系结构，有什么优缺点？
- 2.6 下列哪些指令应在核心态运行：
 - (1) 关中断
 - (2) 读实时时钟
 - (3) 设置实时时钟
 - (4) 改变内存页映射
 - (5) 设置处理器运行状态
- 2.7 你认为下列操作系统采用了哪种体系结构或者具有哪些体系结构的特点：
 - (1) MSDOS
 - (2) Linux
 - (3) Windows 95
 - (4) Windows NT
 - (5) VM/370
 - (6) BSD UNIX
 - (7) Mach
 - (8) Minix
- 2.8 你认为各种操作系统的体系结构分别适合什么样的应用场合？
- 2.9 客户/服务器的操作系统体系结构在分布式系统中使用非常广泛，你认为它能否用于单机环境？Windows 2000/XP具有很多这种体系结构的特征，那么在哪些方面Windows 2000/XP对原有的模型做了哪些调整，你认为这些调整是否有用？
- 2.10 在客户/服务器的体系结构中，进程服务将决定谁被调度。系统的调度机制将在操作系统的核心内完成，而调度的策略则在进程服务中完成。
 - (1) 进程描述表应存放在什么地方？是核心还是进程服务的私有地址空间？
 - (2) 我们知道调度时的切换动作一定是在核心内完成的，那么核心是如何知道切换到哪里去呢？
- 2.11 Windows 2000/XP包括哪些主要的组成部分，各自的作用是什么？
- 2.12 Windows 2000/XP是通过什么方式支持各种不同的硬件平台实现可移植的目标？
- 2.13 Windows 2000/XP的设备驱动程序是做什么的，有哪些类型的设备驱动程序？
- 2.14 环境子系统的主要作用是什么，它是怎样在Windows 2000/XP中发挥它的作用的？
- 2.15 通过进程观察器，你能看见哪些重要的系统进程，它们各自都有什么作用？试着用进程观察器终止一个重要的系统进程（例如WINLOGON），看一看会有什么结果。在DDK中有一个实用程序kill，它可以绕过系统的检查而杀掉这些进程，试着结束SMSS或者CSRSS，看一看有什么结果。

- 2.16 Windows 2000/XP是怎样看待中断和异常的，它提供了什么样的机制去处理它们？
- 2.17 DPC、APC有什么区别？
- 2.18 LPC对RPC做了什么样的优化？
- 2.19 Windows 2000/XP的核心资源管理采用了对象的方式，这有什么好处？在Windows 2000/XP中有哪些类型的对象，都有什么作用？对象管理器是怎样把这些对象组织起来的？
- 2.20 Windows 2000/XP提供了哪些同步互斥的机制？（包括单处理器的情况 and 对称多处理器的情况）。
- 2.21 用Regedit.exe实用程序查看Windows 2000/XP的注册表，了解注册表的构成。
- 2.22 什么是Windows 2000/XP服务，Windows 2000/XP是如何操纵服务的？
- 2.23 请简述WMI的作用和构成情况。
- 2.24 查阅一些有关Linux的资料，看看Linux的体系结构是什么样子的。为了提高性能，Linux在体系结构上做了那些权衡折中。
- 2.25 查阅资料，比较Windows 2000/XP、Windows NT 4、Windows 9x和Windows CE在体系结构上的共同点和差异。想一想为什么会有这样的差异。
- 2.26 使用资源工具包和DDK的工具查看系统状况：① 子系统的启动；② 造成一次系统崩溃，并用调试工具查看故障转储文件（最小转储64K）；③ 窥视核心的非文档化接口；④ 抓一个系统快照，查看当前系统的内存、页表、进程、对象等情况；⑤ 使用性能监视器查看各种不同负载情况下的性能情况；⑥ 研究对象管理器，查看系统对象及其属性；⑦ 查看系统服务活动；⑧ 查看系统的启动日志以及注册表的相关部分，了解Windows 2000/XP的启动机制。
- 2.27 编写一个简单程序（helloworld），使用调试器在汇编级别跟踪它的运行情况。
- 2.28 编写一个最简单的核心态设备驱动程序，安装到你的Windows 2000/XP中，并编写一个用户程序请求该设备驱动程序的功能。试着使用调试工具进行跟踪。调试器可以用Windows 2000/XP的kernel debugger做远程调试或者使用Soft ICE。有关的编程指南可以查看Windows 2000/XP的DDK文档和《Windows Device Drivers》一书。
- 2.29 Windows 2000/XP的服务是要编程实现的，查阅MSDN的相应文章了解这一过程。

第 ③ 章

进程和处理器管理

第 ③ 章

进程和处理器管理

为了描述程序在并发执行时对系统资源的共享，我们需要一个描述程序执行时动态特征的概念，这就是进程。在进程的基础上，引入线程的概念可进一步提高进程的并发性。处理器管理的工作是对处理器资源进行合理的分配使用，以提高处理器利用率，并使各用户公平地得到处理器资源。在本章中，我们将讨论进程和线程的概念、控制和相互关系，以及处理器调度算法。

3.1 进程

进程是处理器管理中的基本概念。本节讨论进程的概念、进程的状态和状态转换。

3.1.1 程序的顺序执行和并发执行

程序的执行可分为顺序执行和并发执行两种方式。顺序执行是指操作系统依次执行各程序，在一个程序的整个执行过程中该程序执行占用所有系统资源，不会中途暂停。顺序执行是单道批处理系统的执行方式，也用于简单的单片机系统。并发执行是指多个程序在一个处理器上的交替执行，这种交替执行在宏观上表现为同时执行。现代操作系统大多采用并发执行方式，具有许多新的特征。引入并发执行的目的是为了提高计算机资源的利用率。

程序的功能是依据指令对输入信息进行处理。程序的顺序执行具有下列三个特征：① 顺序性：程序的执行是按照程序结构所指定的次序进行的，可能的次序有分支、循环或跳转等；② 封闭性：程序在执行过程中独占全部资源，计算机的状态完全由该程序的控制逻辑所决定；③ 可再现性：只要程序执行的初始条件相同，执行结果就完全相同。例如，在程序中可利用空指令控制时间关系。

程序的并发执行可提高计算机资源的利用率，但并发执行也改变了程序的执行环境，并导致一些在顺序执行方式下可正常工作的程序在并发执行方式下不能正常工作。这种程序执行环境的变化体现在以下三个方面：① 间断(异步)性：处理器交替执行多个程序，每个程序都是以“走走停停”的方式执行，可能走到中途停下来，而且程序无法预知每次执行和暂停的时间长度，从而失去了原有的时序关系；② 失去封闭性：由于多个程序共享一个计算机系统的多种资源，因此每个程序的执行都会受其他程序的控制逻辑的影响，例如，一个程序写到存储器中的数据可能被另一个程序修改，失去原有的数据不变特征；③ 失去可再现性：由于程序执行环境的封闭性不再成立，因此程序每次执行的环境可能会不同，执行环境在程序的两次执行期间发生变化导致执行结果的不同，程序的执行结果失去原有的可重复特征。

程序执行是为了对输入信息进行处理,并得到相应的处理结果。为此,程序在并发执行时必须保持封闭性和可再现性。由于并发执行失去封闭性的原因是共享资源的影响,因此现在的工作就是去掉这种影响。

1966年, Bernstein给出了程序并发执行的条件。注意:他所给出的条件并没有考虑执行速度的影响。假设程序 $P(i)$ 所访问的共享变量的读集和写集分别为 $R(i)$ 和 $W(i)$,则任意两个程序 $P(i)$ 和 $P(j)$ 可以并发执行的条件有以下三条:

- 1) $R(i) \cap W(j) = \Phi$ 。
- 2) $W(i) \cap R(j) = \Phi$ 。
- 3) $W(i) \cap W(j) = \Phi$ 。

其中,前两个条件保证一个程序在两次读操作之间存储器中的数据不会发生变化;最后一个条件保证程序的写操作的结果不会丢失。只要同时满足三个条件,并发执行的程序就可保持封闭性和可再现性。但这并没有解决所有问题,在实际的程序执行过程中很难对这三个条件进行检查。下面我们讨论如何通过操作系统的有效管理来保证并发执行程序的封闭性和可再现性。

3.1.2 进程的定义和描述

进程(process)是一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。进程与处理器、存储器和外设等资源的分配和回收相对应,进程是计算机系统资源的使用主体。在操作系统中引入进程的并发执行,是指多个进程在同一计算机操作系统中的并发执行。引入进程并发执行可提高对硬件资源的利用率,但又带来额外的空间和时间开销,增加了操作系统的复杂性。

作为描述程序执行过程的概念,进程具有动态性、独立性、并发性和结构化等特征。动态性是指进程具有动态的地址空间,地址空间的大小和内容都是动态变化的。地址空间的内容包括代码(指令执行和处理器状态的改变)、数据(变量的生成和赋值)和系统控制信息(进程控制块的生成和删除)。独立性是指各进程的地址空间相互独立,除非采用进程间通信手段,否则不能相互影响。并发性也称为异步性,是指从宏观上看,各进程是同时独立运行的。结构化是指进程地址空间的结构划分,如代码段、数据段和核心段划分。

进程和程序是两个密切相关的不同概念,它们在以下几个方面存在区别和联系。

- 进程是动态的,程序是静态的。程序是有序代码的集合;进程是程序的执行。进程通常不可以在计算机之间迁移;而程序通常对应着文件、静态和可以复制。
- 进程是暂时的,程序是永久的。进程是一个状态变化的过程;程序可长久保存。
- 进程与程序的组成不同:进程的组成包括程序、数据和进程控制块(即进程状态信息)。
- 进程与程序是密切相关的。通过多次执行,一个程序可对应多个进程;通过调用关系,一个进程可包括多个程序。进程可创建其他进程,而程序并不能形成新的程序。

进程是程序代码的执行过程,但并不是所有代码执行过程都从属于某个进程。例如,处理器调度器是操作系统中的一段代码,它完成的功能包括:①把处理器从一个进程切换到另一个进程;②防止某进程独占处理器。处理器调度器的执行过程就不与进程相对应。

进程控制块 (Process Control Block, PCB) 是由操作系统维护的用来记录进程相关信息的数据结构。每个进程在操作系统中都有对应的进程控制块, 操作系统维护的进程控制块总数可能会有限制。操作系统依据进程控制块对进程进行控制和管理, 进程控制块中的内容会随进程推进而动态改变。进程控制块处于操作系统核心, 通常不能由应用程序自身的代码来直接访问, 而要通过系统调用进行访问。在UNIX中也可通过进程文件系统(/proc)直接访问进程映像。

进程控制块的内容可分成进程描述信息、进程控制信息、资源占用信息和处理器现场保护结构这4个部分。进程描述信息包括进程标识符(process ID)、进程名 (通常是可执行文件名)、用户标识符(user ID)和进程组(process group)等。进程控制信息包括当前状态、优先级、代码执行入口地址、程序的外存地址、运行统计信息 (执行时间、页面调度)、进程阻塞原因等。资源占用信息是指进程占用的系统资源列表。处理器现场保护结构保存寄存器值, 如通用寄存器、程序计数器PC、状态字PSW, 地址包括栈指针等。

操作系统要将处于同一状态的进程的进程控制块组织在一起, 常用的组织方式有链表和索引表两种。链表方式是将同一状态的进程控制块组成一个链表, 多个状态对应多个不同的链表, 如就绪链表和阻塞链表等。索引表方式是将同一状态的进程归入一个索引表, 再由索引指向相应的进程控制块, 多个状态对应多个不同的索引表, 如就绪索引表和阻塞索引表等。

对进程执行活动全过程的静态描述称为进程上下文。进程上下文包括进程的用户地址空间内容、处理器中寄存器内容及与该进程相关的核心数据结构等, 可分成用户级上下文、寄存器级上下文和系统级上下文。用户级上下文是指进程的用户地址空间, 包括用户正文段、用户数据段和用户栈。寄存器级上下文是指程序寄存器、处理器状态寄存器、栈指针、通用寄存器的值等。系统级上下文包括进程的静态部分 (PCB和资源表格) 和由核心栈等构成的动态部分。

3.1.3 进程的状态转换

进程在从创建到终止的全过程中一直处于一个不断变化的过程。为了刻画进程的这个变化过程, 所有操作系统都把进程分成若干种状态, 约定各种状态间的转换条件。对进程状态的刻画也经历了一个不断精确化的过程。下面我们就讨论进程的状态模型。

1. 五状态进程模型

在五状态进程模型中, 进程状态被分成下列五种状态。进程在运行过程中主要是在就绪、运行和阻塞三种状态间进行转换。创建状态和退出状态描述进程创建的过程和进程退出的过程。如图3-1所示。

1) 运行状态(Running): 进程占用处理器资源; 处于此状态的进程的数目小于等于处理器的数目。在没有其他进程可以执行时 (如所有进程都在阻塞状态), 通常会自动执行系统的空闲进程。

2) 就绪状态(Ready): 进程已获得除处理器外的所需资源, 等待分配处理器资源; 只要分配了处理器进程就可执行。就绪进程可以按多个优先级来划分队列。例如, 当一个进程由于时间片用完而进入就绪状态时, 排入低优先级队列; 当进程由于I/O操作完成而进入就绪状态时, 排入高优先级队列。

3) 阻塞状态(Blocked): 当进程由于等待I/O操作或进程同步等条件而暂停运行时, 它处于阻

塞状态。在条件满足之前，即使把处理器分配给该进程，它也是无法继续执行的。

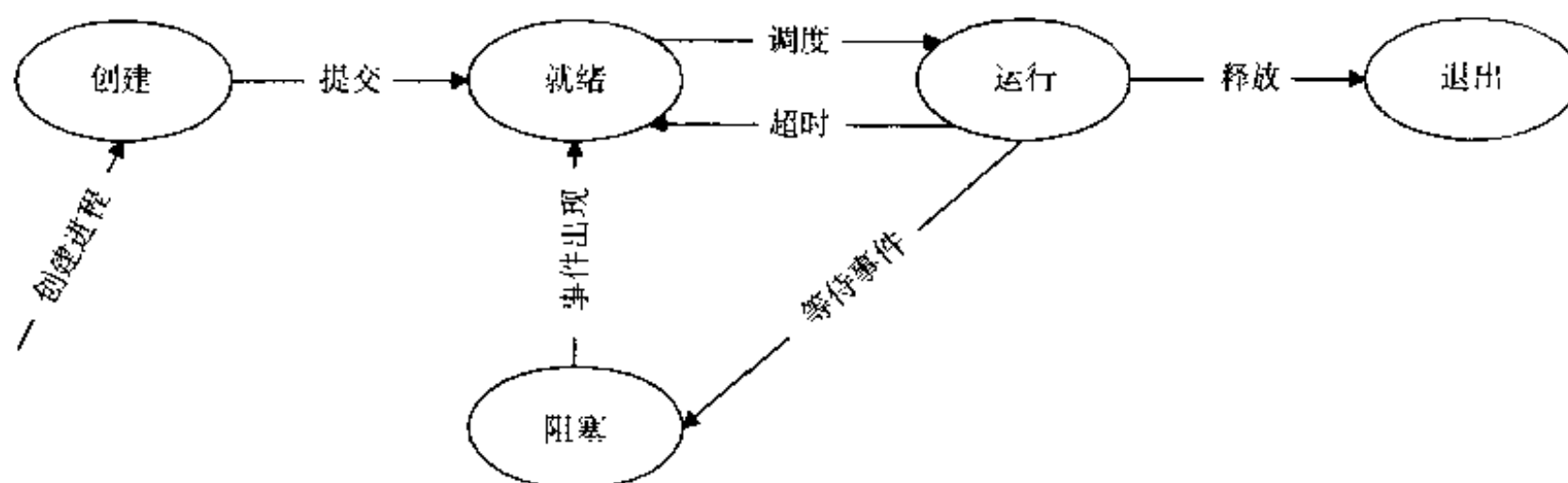


图3-1 五状态进程模型(状态变迁)

4) 创建状态(New): 进程正在创建过程中，还不能运行。操作系统在创建状态要进行的工作包括分配和建立进程控制块表项、建立资源表格（如打开文件表）并分配资源、加载程序并建立地址空间表等。

5) 退出状态(Exit): 进程已结束运行，回收除进程控制块之外的其他资源，并让其他进程从进程控制块中收集有关信息（如记帐和将退出代码传递给父进程）。

五状态进程模型中的状态转换主要包括下列几种。操作系统中多个进程的并发执行是通过调度与超时两种转换间的循环，或调度、等待事件和事件出现三种转换间的循环来描述的。

1) 创建新进程: 创建一个新进程，以运行一个程序。创建新进程的可能原因包括用户登录、操作系统创建以提供某项服务、批处理作业等。

2) 收容(Admit, 也称为提交): 收容一个新进程，进入就绪状态。由于性能、内存、进程总数等原因，系统会限制并发进程总数。

3) 调度运行(Dispatch): 从就绪进程表中选择一个进程，进入运行状态。

4) 释放(Release): 由于进程完成或失败而终止进程运行，进入结束状态。为了简洁，状态变迁图中只画出了运行状态到退出状态间的释放转换；但实际上，还存在从就绪状态或阻塞状态到退出状态的释放转换。运行到结束的转换可分为正常退出(Exit)和异常退出(abort)；其中异常退出是指进程执行超时、内存不够、非法指令或地址访问、I/O操作失败、被其他进程所终止等原因而退出。从就绪状态或阻塞状态到结束状态的释放转换可能是由于多种原因引发，如父进程可在任何时间终止子进程。

5) 超时 (Timeout): 由于用完时间片或高优先级进程就绪等原因导致进程暂停运行。

6) 事件等待 (Event Wait): 进程要求的事件未出现而进入阻塞；可能的原因包括：申请系统服务或资源、通信、I/O操作等。

7) 事件出现 (Event Occurs): 进程等待的事件出现；如操作完成、申请成功等。

对于五状态进程模型，操作系统要解决的一个重要问题是当一个事件出现时如何检查阻塞进程表中的进程状态，如图3-2所示。当进程多时，这对系统性能影响很大。如图3-3所示，一种可能的做法是按等待事件类型，排成多个队列。

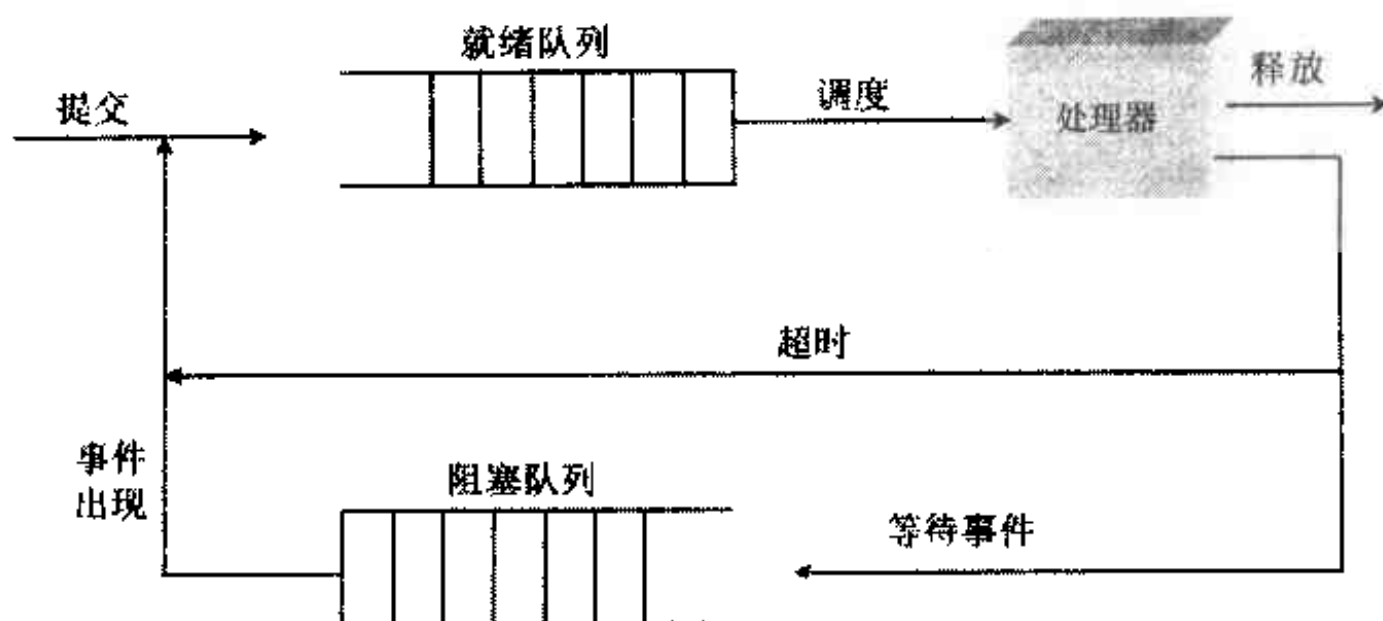


图3-2 五状态进程模型(单队列结构)

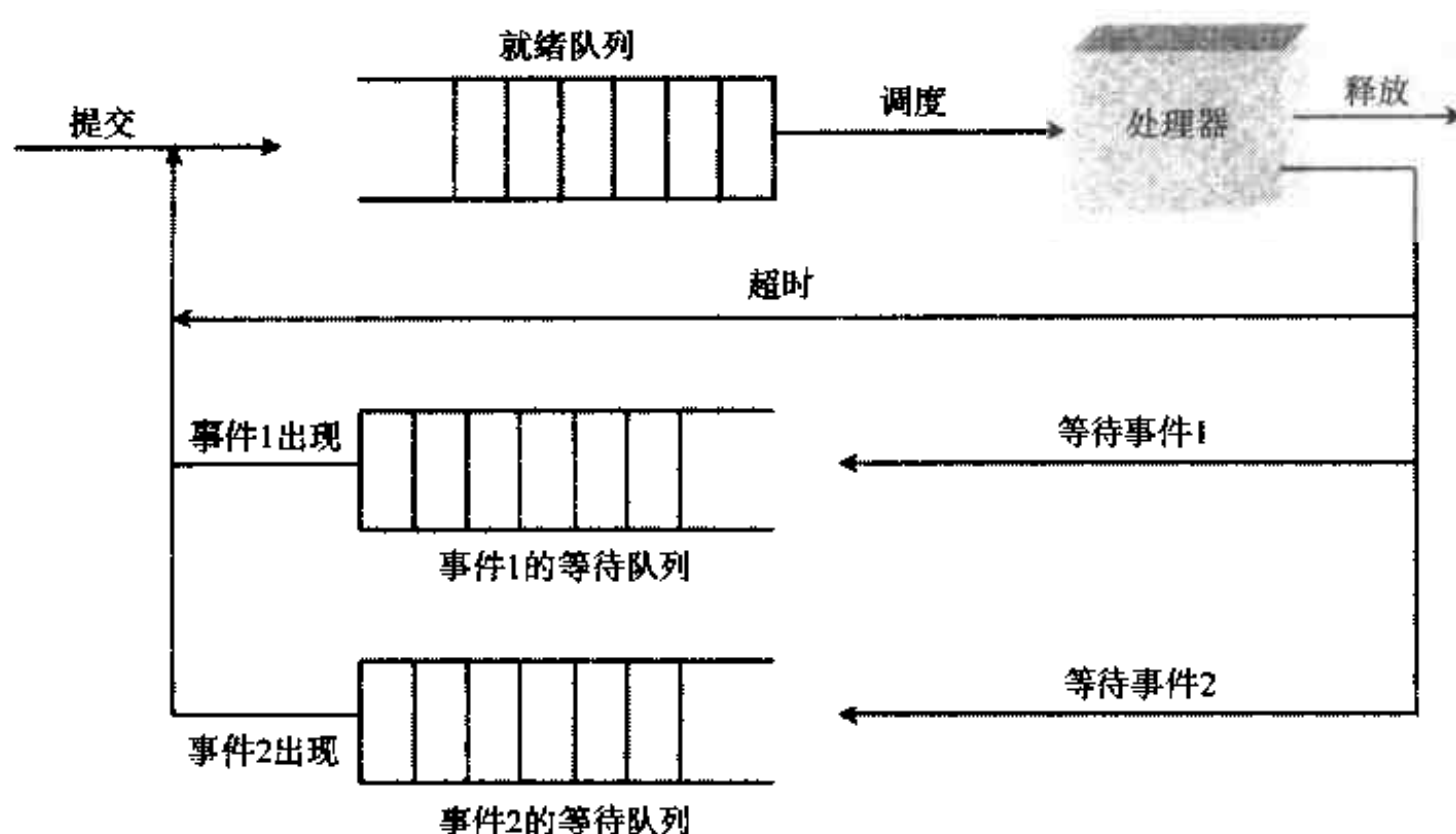


图3-3 五状态进程模型(多队列结构)

2. 挂起进程模型

五状态进程模型没有区分进程地址空间位于内存还是外存，而在操作系统中引入虚拟存储管理技术后，需要进一步区分进程的地址空间状态。这个问题的出现是由于进程优先级的引入，一些低优先级进程可能等待较长时间，从而被对换至外存。这种做法可得到下列的好处：① 提高处理器效率：就绪进程表为空时，有空闲内存空间用于提交新进程，可提高处理器效率；② 可为运行进程提供足够内存：资源紧张时，可把某些进程对换至外存；③ 有利于调试：在调试时，挂起被调试进程，可方便对其地址空间进行读写。

与五状态进程模型相比，挂起进程模型把原来的就绪状态和阻塞状态进行了细分。在单挂起进程模型中增加了一个阻塞挂起状态（见图3-4）；而在双挂起进程模型中增加了就绪挂起和阻塞挂起两个状态（见图3-5）。这时，原来的就绪状态和阻塞状态的意义也发生了一些变化。下面

列出的是在挂起进程模型中四种意义有变化的状态或新的状态。

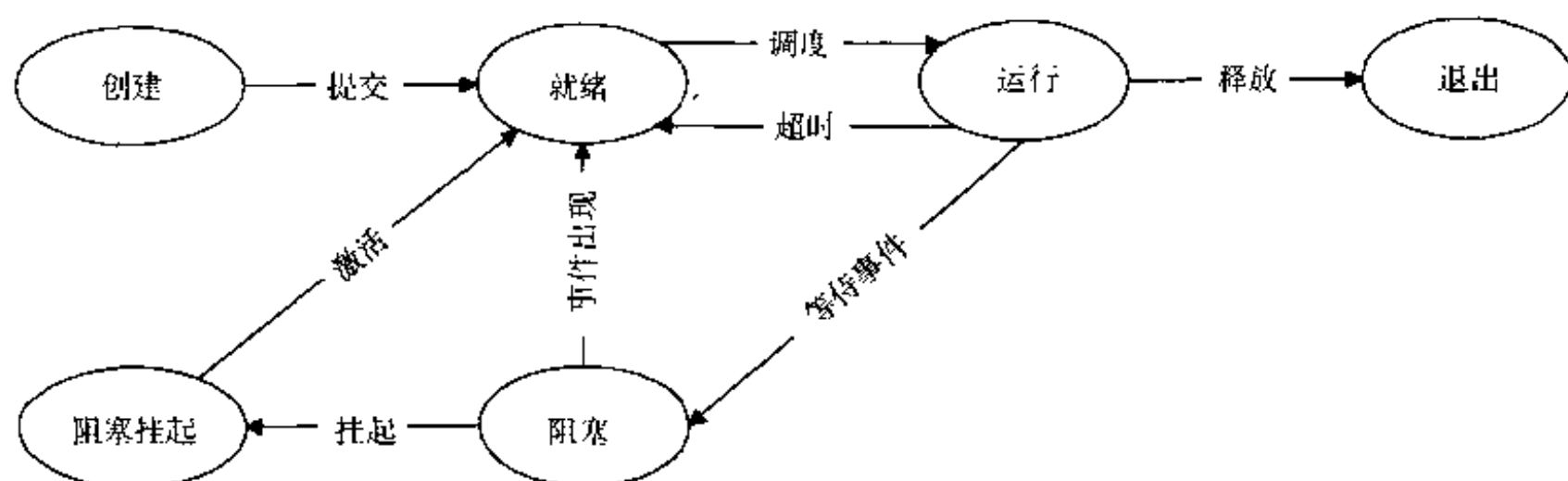


图3-4 单挂起进程模型

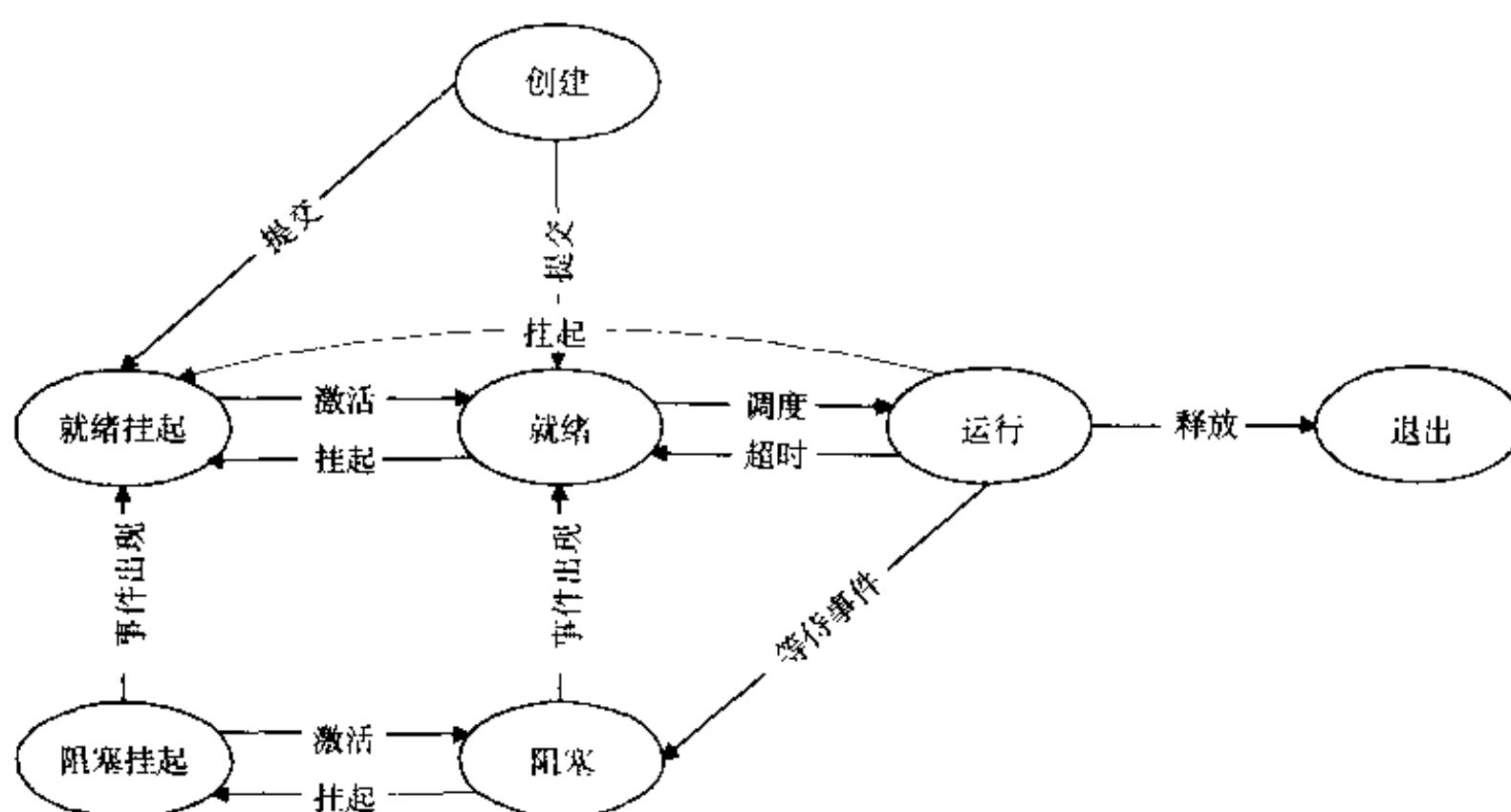


图3-5 双挂起进程模型

- 1) 就绪状态(ready): 进程在内存且可立即进入运行状态。
- 2) 阻塞状态(blocked): 进程在内存并等待某事件的出现。
- 3) 阻塞挂起状态 (blocked, suspend): 进程在外存并等待某事件的出现。
- 4) 就绪挂起状态 (ready, suspend): 进程在外存，但只要进入内存，即可运行。

在挂起进程模型中，新引入的状态转换有挂起和激活两类，意义有变化的状态转换有事件出现和收容两类。

- 1) 挂起 (suspend): 把一个进程从内存转到外存；可能有以下几种情况：

- 阻塞到阻塞挂起：没有进程处于就绪状态或就绪进程要求更多内存资源时，会进行这种转换，以提交新进程或运行就绪进程。
- 就绪到就绪挂起：当有高优先级阻塞（系统认为会很快就绪的）进程和低优先级就绪进

程时，系统会选择挂起低优先级就绪进程。

- 运行到就绪挂起：对抢先式分时系统，当有高优先级阻塞挂起进程因事件出现而进入就绪挂起时，系统可能会把运行进程转到就绪挂起状态。

2) 激活 (activate)：把一个进程从外存转到内存，可能有以下几种情况：

- 就绪挂起到就绪：就绪挂起进程优先级高于就绪进程或没有就绪进程时，会进行这种转换。
- 阻塞挂起到阻塞：当一个进程释放足够内存时，系统会把一个高优先级阻塞挂起进程激活，系统认为会很快出现该进程所等待的事件。

3) 事件出现 (event occur)：进程等待的事件出现，如操作完成、申请成功等；可能的情况有：

- 阻塞到就绪：针对内存进程的事件出现。
- 阻塞挂起到就绪挂起：针对外存进程的事件出现。

4) 收容(admit)：收容一个新进程，进入就绪状态或就绪挂起状态。进入就绪挂起的原因是系统希望保持一个大的就绪进程表（挂起和非挂起）。

我们这里讨论的进程状态模型是对实际操作系统中所使用的进程状态定义的抽象和简化，我们将在操作系统实例中介绍它们与实际操作系统中的进程状态之间的关系。

3.2 进程控制

操作系统对进程的控制是依据用户命令和系统状态来决定的。进程控制的功能是完成进程状态的转换。用户可在一定程序上对进程的状态进行控制。这种控制主要体现在进程的创建与退出，以及进程的挂起与激活。

3.2.1 进程的创建和退出

在操作系统完成初始化后，系统就可创建进程。在进程的创建过程中，操作系统要进行进程控制块等相关数据结构的维护。一个进程可利用系统调用功能来创建新的进程，创建者称为父进程，而被创建的新进程称为子进程。按子进程是否覆盖父进程和是否加载新程序，子进程的创建可分为如表3-1中所示的fork、spawn和exec三种类型。由于复制现有进程的上下文时一定会产生新进程，所以只有三种类型。

表 3-1

	产生新进程	不产生新进程
复制现有进程的上下文	fork (新进程的系统上下文会有不同)	-
加载程序	spawn (创建新进程并加载新程序)	exec (加载新程序并覆盖自身)

子进程与父进程存在密切的关系，子进程的许多属性就是从父进程继承来的；与此同时，子进程又与父进程有区别，形成自己独立的属性。子进程可以从父进程中继承的属性包括：用户标识符、环境变量、打开文件、文件系统的当前目录、控制终端、已经连接的共享存储区和信号处

理例程入口表等。子进程不能从父进程继承的属性包括：进程标识符和父进程标识符等。还有一些属性可在进程创建中约定是否能从父进程继承。各种操作系统都有相应的系统调用完成进程创建，主要工作过程都是一致的，但它们也会在细节上有一些区别。

进程的退出是通过相应的系统调用进行的，也称为“进程终止”。例如，在C语言中可调用exit()来终止进程。在进程的退出过程中，操作系统要删除系统维护的相关数据结构并回收进程占用的系统资源，例如，释放进程占用的内外存空间、关闭所有打开文件、释放共享内存段和解除各种锁定等。

3.2.2 进程的阻塞和唤醒

进程在执行过程中会因为等待I/O操作完成或等待某个事件出现而进入阻塞状态。当处于阻塞状态的进程所等待的操作完成或事件出现时，进程将会从阻塞状态唤醒而进入就绪状态。用户可通过相应系统调用来等待某个事件或唤醒某个阻塞进程。

UNIX系统中，与进入阻塞状态相关的系统调用主要有：暂停一段时间(sleep)、暂停并等待信号(pause)和等待子进程暂停或终止(wait)。与唤醒阻塞进程相关的系统调用主要是发送信号到某个或一组进程(kill)。各系统调用的简要描述如下：

1) sleep将在指定的时间seconds内挂起本进程。其调用格式为：“unsigned sleep(unsigned seconds);”，返回值为实际的挂起时间。

2) pause挂起本进程以等待信号，接收到信号后恢复执行。当接收到终止进程信号时，该调用不再返回。其调用格式为“int pause(void);”。

3) wait挂起本进程以等待子进程的结束，子进程结束时返回。当父进程创建多个子进程且已有子进程退出时，父进程中wait函数在第一个子进程结束时返回。其调用格式为“pid_t wait(int *stat_loc);”，返回值为子进程ID。

4) kill可发送信号sig到某个或一组进程pid。其调用格式为：“int kill(pid_t pid, int sig);”。信号的定义在文件“/usr/include/sys/signal.h”中。命令“kill”可用于向进程发送信号。例如，“kill -9 100”将发送SIGKILL到ID为100的进程；该命令将终止该进程的执行。

在Windows NT和Windows 2000/XP中，处理器的调度对象为线程，用户可通过系统调用SuspendThread和ResumeThread来挂起或激活线程。一个线程可被多次挂起和多次激活。在线程控制块中有一个挂起计数(suspend count)，挂起操作使该计数加1，激活操作使该计数减1。当挂起计数从0变为1时，线程进入阻塞状态；当挂起计数由1变为0时，线程恢复执行。各系统调用的简要描述如下：

1) 挂起：Windows NT中的SuspendThread可挂起指定的线程。

```
DWORD SuspendThread( HANDLE hThread // 线程句柄
);
```

2) 激活：Windows NT中的ResumeThread可恢复指定线程的执行。

```
DWORD ResumeThread( HANDLE hThread // 表示被恢复的线程
);
```


3.2.3 Windows 2000/XP进程管理

Windows 2000/XP中的进程是系统资源分配的基本单位。Windows 2000/XP进程是作为对象来管理的，可通过相应句柄(handle)来引用进程对象，操作系统提供一组控制进程对象的服务(services)。进程对象的属性包括：进程标识(PID)、资源访问令牌(Access Token)、进程的基本优先级(Base Priority)和默认亲和处理器集合(Processor Affinity)等。

为了支持Win32、OS/2、POSIX等多种运行环境子系统，Windows 2000/XP核心的进程之间没有任何关系（包括父子关系），各运行环境子系统分别建立、维护和表达各自的进程关系。例如，POSIX环境子系统维护POSIX应用进程间的父子关系。如图3-6所示，Windows NT和Windows 2000/XP把Win32环境子系统设计成整个系统的主子系统，一些基本的进程管理功能被放置在Win32子系统中，POSIX和OS/2等其他的子系统会利用Win32子系统的功能来实现自身的功能。在Windows 2000/XP中，与一个运行环境子系统中的应用进程相关的进程控制块信息会分布在本运行环境子系统、Win32子系统和系统内核中。

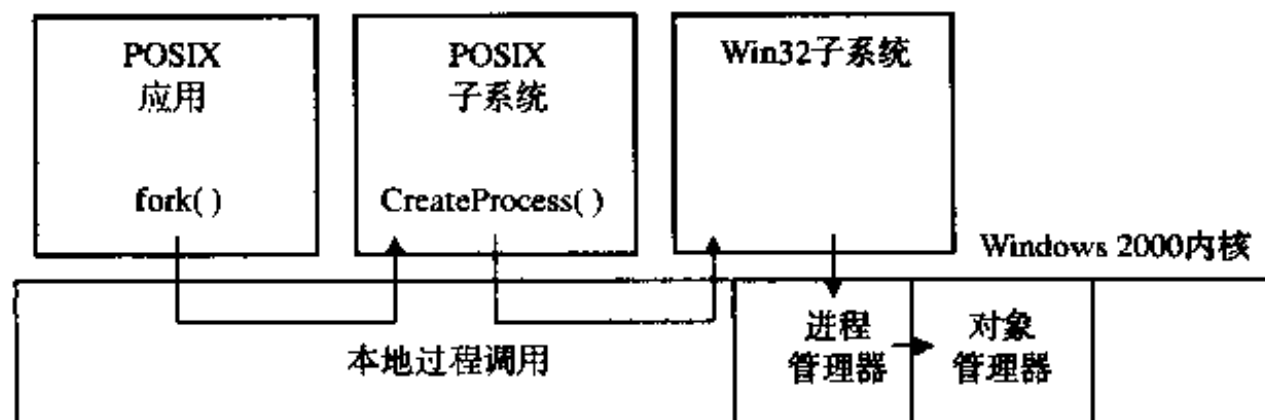


图3-6 Windows 2000/XP的进程关系

如图3-7所示，Windows 2000/XP中的每个Win32进程都由一个执行体进程块(EPROCESS)表示，执行体进程块描述进程的基本信息，并指向其他与进程控制相关的数据结构。执行体进程块中的主要内容包括：① 线程块列表：描述属于该进程的所有线程的相关信息，以便线程调度器进行处理器资源的分配和回收；② 虚拟地址空间描述表(Virtual Address space Descriptor, VAD)：描述进程地址空间各部分属性，用于虚拟存储管理；③ 对象句柄列表：当进程创建或打开一个对象时，就会得到一个代表该对象的句柄，用于对象访问，对象句柄列表维护该进程正在访问的所有对象列表。

Windows 2000/XP支持各环境子系统都有相应的系统调用实现进程控制。Win32子系统的进程控制系统调用主要有CreateProcess、ExitProcess和TerminateProcess，系统调用CreateProcess用于子进程创建，而ExitProcess和TerminateProcess用于子进程退出。这几个系统调用的简要介绍如下：

1) CreateProcess创建新进程及其主线程，以执行指定的程序。Win32进程在创建时可指定从父进程继承的属性，许多对象句柄的继承特征可在对象创建或打开时指定，从而影响新进程的执行。新进程可以继承的进程属性包括：打开文件的句柄、各种对象（如进程、线程、信号量、管道等）的句柄、环境变量、当前目录、原进程的控制台、原进程的进程组标识符等。新进程不能从父进程继承的属性包括：优先权类、内存句柄、DLL模块句柄等。

2) ExitProcess和TerminateProcess都可用于进程退出，它们会终止调用进程内的所有线程。

这两个系统调用的区别在于终止操作是否完整。ExitProcess终止一个进程和它的所有线程；它的终止操作是完整的，包括关闭所有对象句柄、所有线程等。TerminateProcess终止指定的进程和它的所有线程；它的终止操作是不完整的，通常只用于异常情况下对进程的终止。

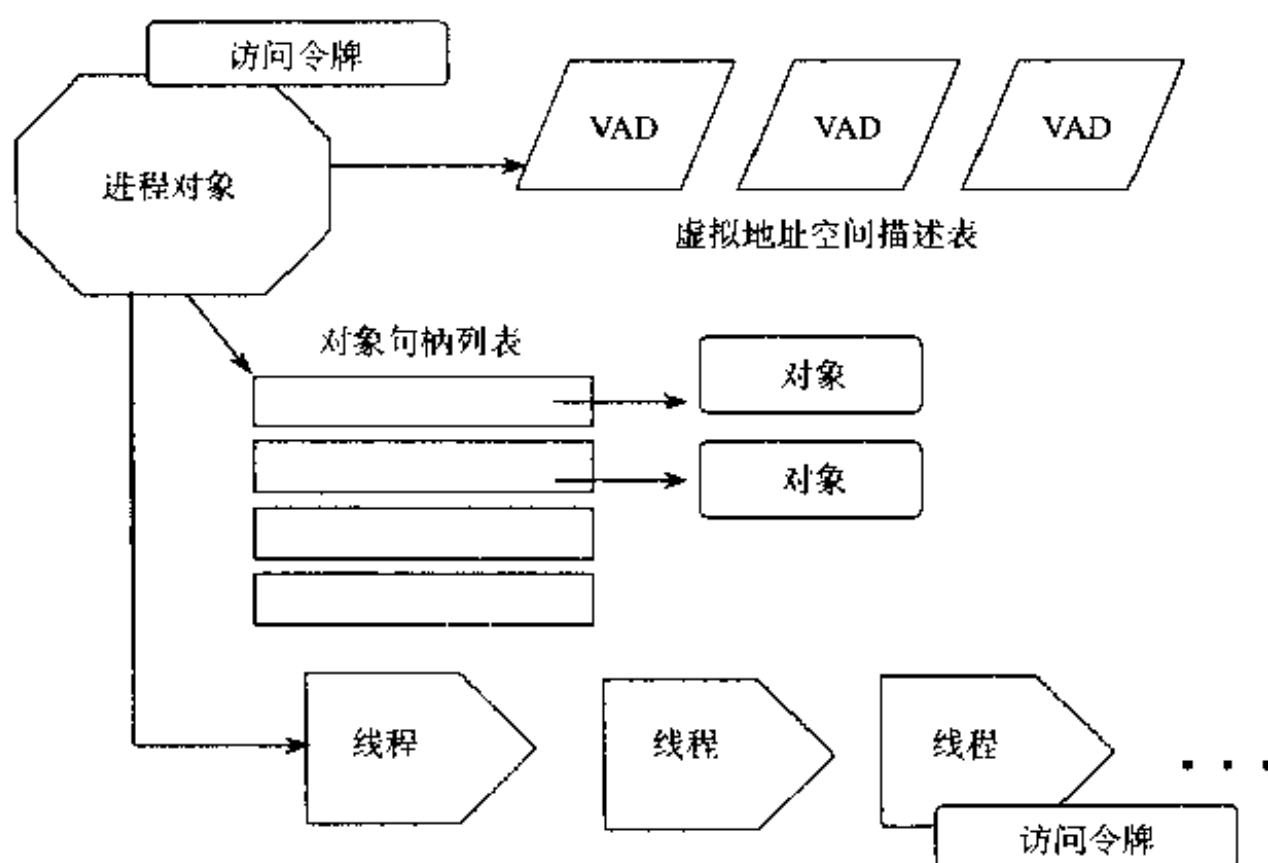


图3-7 Windows 2000/XP中的Win32进程结构

Windows 2000/XP提供一整套机制用于程序调试。在操作系统维护的进程对象属性中包括一个用于调试时进程间通信的通道，通过此通道可了解和控制被调试进程的运行，访问被调试进程地址空间的内容。调试器进程（debugger）在创建被调试进程（target）时可指定DEBUG_PROCESS标志或利用DebugActiveProcess函数，在调试器与被调试进程间建立调试关系，被调试进程会向调试器通报所有调试事件。被调试进程向调试器进程发送的调试事件包括：创建新进程、新线程、加载DLL、执行断点等。调试器可通过WaitForDebugEvent和ContinueDebugEvent来等待被调试进程的调试事件和继续调试进程的运行。WaitForDebugEvent可在指定的时间内等待可能的调试事件；ContinueDebugEvent可使被调试事件暂停的进程继续运行。调试器进程可通过ReadProcessMemory()和WriteProcessMemory()来读写被调试进程的存储空间。

3.3 线程

在操作系统中，进程的引入提高了计算机资源的利用效率。但在进一步提高进程的并发性时，人们发现进程切换开销占的比重越来越大，同时进程间通信的效率也受到限制。线程的引入正是为了简化线程间的通信，以小的开销来提高进程内的并发程度。本节讨论线程的概念和它与进程的差异。

3.3.1 线程的概念

在只有进程概念的操作系统中，进程是存储器、外设等资源的分配单位，同时也是处理器调

度的对象。为了提高进程内的并发性，在引入线程的操作系统中，把线程作为处理器调度的对象，而把进程作为资源分配单位，一个进程内可同时有多个并发执行的线程。

线程(Thread)是一个动态的对象，它是处理器调度的基本单位，表示进程中的一个控制点，执行一系列的指令。由于同一进程内各线程都可访问整个进程的所有资源，因此它们之间的通信比进程间通信要方便；而同一进程内的线程间切换也会由于许多上下文的相同而简化。如图3-8所示，线程与进程是两个密切相关的概念。我们可以把原来的进程概念理解为只有一个主线程的进程。

同一进程内各线程的差异主要体现在线程状态、寄存器上下文和堆栈等必不可少的线程执行环境上。这样，在操作系统中引入线程概念，就可减小并发执行的时间和空间开销，容许通过在系统中建立更多的线程来提高并发性。线程的优点具体体现在以下几方面：① 线程的创建时间比进程短；② 线程的终止时间比进程短；③ 同进程内的线程切换时间比进程短；④ 由于同进程内线程间共享内存和文件资源，因此可直接进行不通过内核的通信。

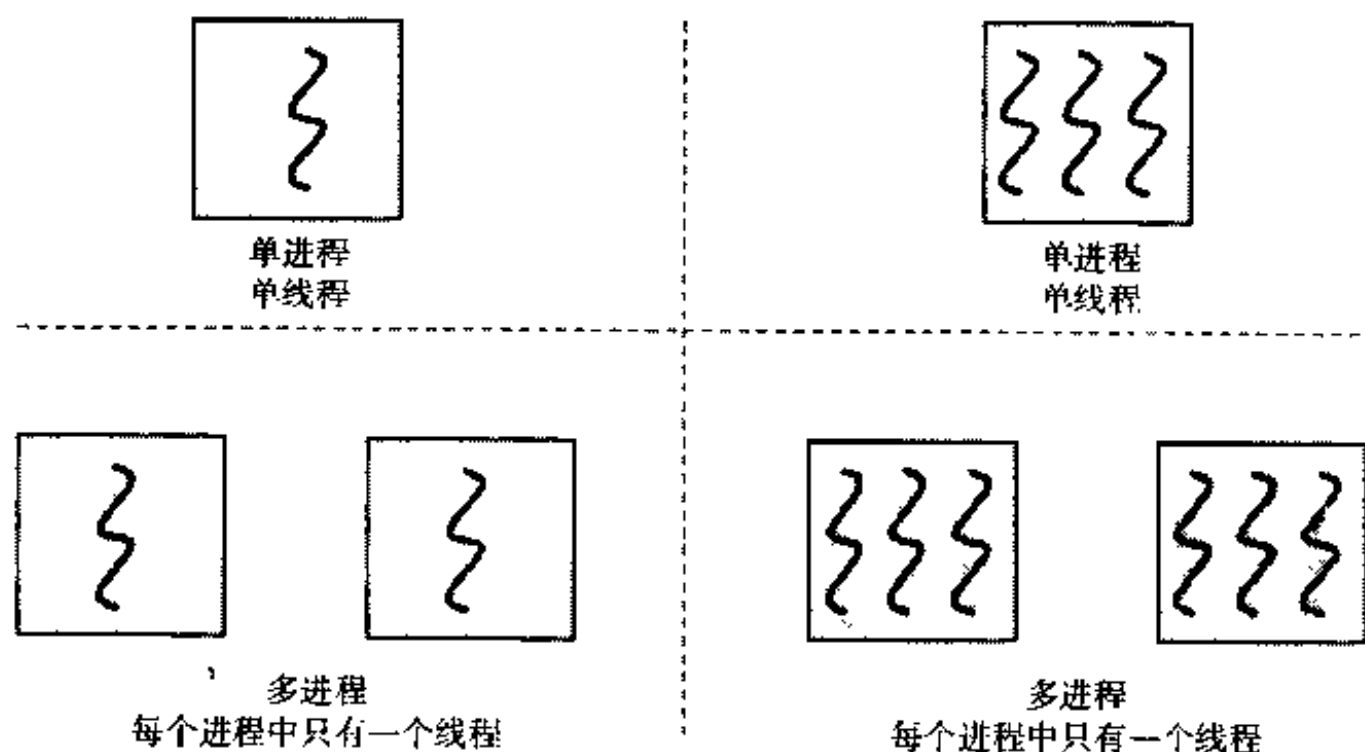


图3-8 进程与线程的关系

在操作系统中有多种方式可实现对线程的支持。最自然的方法是由操作系统内核提供线程的控制机制。在只有进程概念的操作系统中可由用户程序利用函数库提供线程的控制机制。还有一种做法是同时在操作系统内核和用户程序两个层次上提供线程控制机制。这就构成了内核线程、用户线程和轻量级进程这三种线程的实现方式。

内核线程(kernel-level thread)是指由操作系统内核完成创建和撤销，用来执行一个指定的函数线程。在支持内核线程的操作系统中，内核维护进程和线程的上下文信息以及线程切换由内核完成。一个内核线程由于I/O操作而阻塞，不会影响其他线程的运行。这时处理器时间片分配的对象是线程，所以多线程的进程获得更多处理器时间。Windows NT和Windows 2000/XP支持内核线程。

用户线程(user-level thread)是指不依赖于操作系统核心，由应用进程利用线程库提供创建、

同步、调度和管理线程的函数来控制的线程。由于用户线程的维护由应用进程完成，不需要操作系统内核了解用户线程的存在，因此可用于不支持内核线程的多进程操作系统，甚至是单用户操作系统。用户线程切换不需要内核特权，用户线程调度算法可针对应用优化。在许多应用软件中都有自己的用户线程。例如数据库系统Informix和图形处理软件Aldus PageMaker等。由于用户线程的调度在应用软件内部进行，通常采用非抢先式和更简单的规则，也无需用户态/核心态切换，因此速度特别快。当然，由于操作系统内核不了解用户线程的存在，当一个线程在进入系统调用后阻塞时，整个进程都必须等待。这时处理器时间片是分配给进程的，进程内有多个线程时，每个线程的执行时间相对就少。

轻量级进程(LightWeight Process)是指由内核支持的用户线程。一个进程可有一个或多个轻量级进程，每个轻量级进程由一个单独的内核线程来支持。由于同时提供内核线程控制机制和用户线程库，因此可很好地把内核线程和用户线程的优点结合起来。

3.3.2 进程和线程的比较

由于进程与线程的密切相关，因此我们有必要比较一下进程与线程的差异（见图3-9）。可从以下三个角度来比较进程和线程的不同。① 地址空间资源：不同进程的地址空间是相互独立的，而同一进程的各线程共享同一地址空间。② 通信关系：进程间通信必须使用操作系统提供的进程间通信机制，而同一进程中的各线程间可以通过直接读写进程数据段（如全局变量）来进行通信。当然同一进程中各线程间的通信也需要同步和互斥手段的辅助，以保证数据的一致性。③ 调度切换：同一进程中的线程上下文切换比进程上下文切换要快得多。

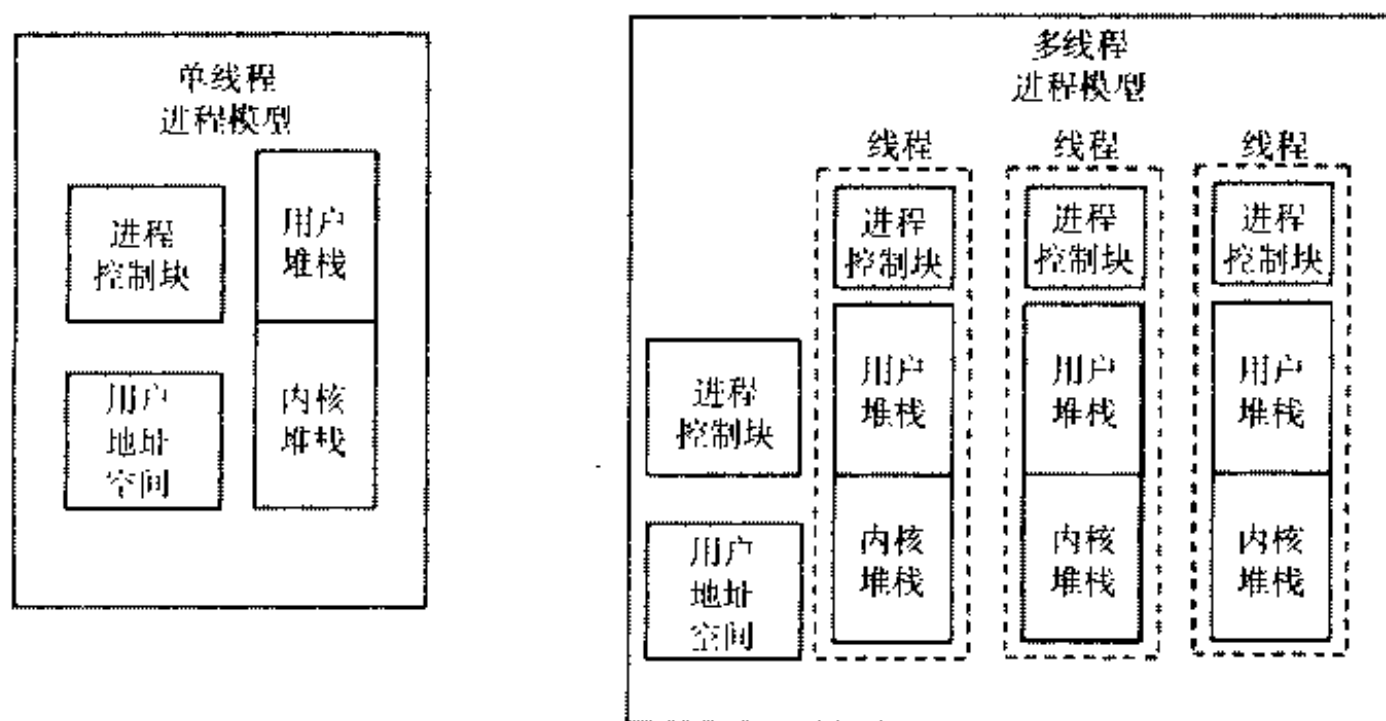


图3-9 进程和线程的比较

3.3.3 Windows 2000/XP线程

Windows 2000/XP的线程是内核线程，系统的处理器调度对象为线程。线程上下文主要包括

寄存器、线程环境块、核心栈和用户栈。Windows 2000/XP把线程状态分成下面七种，如图3-10所示，与单挂起进程模型很相似，它们的主要区别在于从就绪状态到运行状态的转换中间多了一个备用状态，以优化线程的抢先特征。

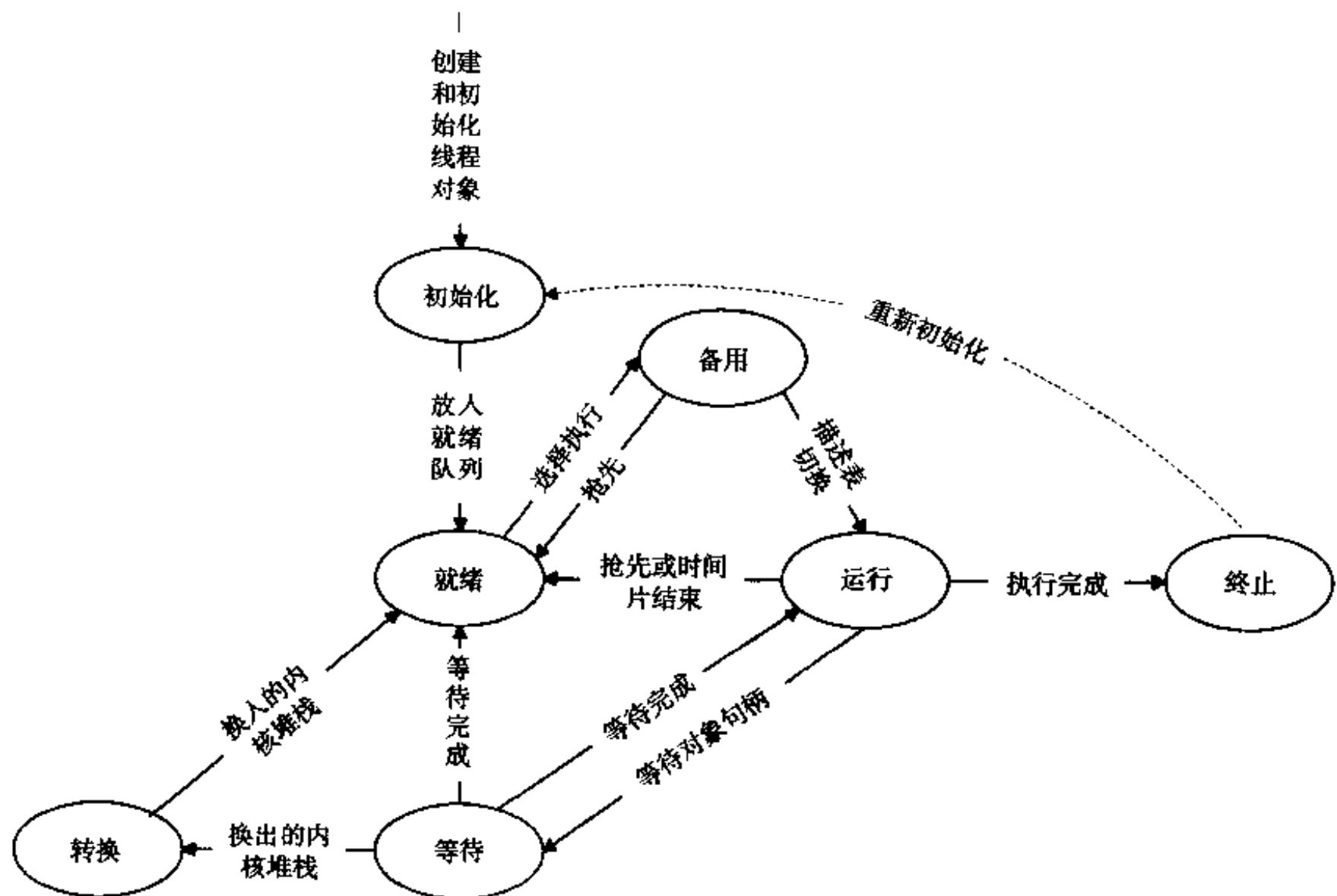


图3-10 Windows 2000/XP的线程状态

- 1) 就绪状态(ready): 线程已获得除处理器外的所需资源，正等待调度执行。
- 2) 备用状态(standby): 已选择好线程的执行处理器，正等待描述表切换，以进入运行状态。系统中每个处理器上只能有一个处于备用状态的线程。
- 3) 运行状态(running): 已完成描述表切换，线程进入运行状态。线程会一直处于运行状态，直到被抢先、时间片用完、线程终止或进入等待状态。
- 4) 等待状态(waiting): 线程正等待某对象，以同步线程的执行。当等待事件出现时，等待结束，并根据优先级进入运行或就绪状态。
- 5) 转换状态(transition): 转换状态与就绪状态类似，但线程的内核堆栈位于外存。当线程等待事件出现而它的内核堆栈处于外存时，线程进入转换状态；当线程内核堆栈被调回内存时，线程进入就绪状态。
- 6) 终止状态(terminated): 线程执行完就进入终止状态；如执行体有一个指向线程对象的指针，可将处于终止状态的线程对象重新初始化，并再次使用。

7) 初始化状态(Initialized): 线程创建过程中的线程状态。

Windows 2000/XP有一组相关的系统调用用于线程控制。CreateThread完成线程创建, 在调用进程的地址空间上创建一个线程, 以执行指定的函数; 它的返回值为所创建线程的句柄。ExitThread用于结束当前线程。SuspendThread可挂起指定的线程。ResumeThread可激活指定线程, 它的对应操作是递减指定线程的挂起计数, 当挂起计数减为0时, 线程恢复执行。

3.4 进程互斥和同步

由于多进程在操作系统中的并发执行, 它们之间存在着相互制约的关系。这就是进程间的同步和互斥关系。进程同步是指多个进程中发生的事件存在某种时序关系, 必须协同动作、相互配合, 以共同完成一个任务。进程互斥是指由于共享资源所要求的排它性, 进程间要相互竞争, 以使用这些互斥资源。下面讨论如何实现进程同步和互斥。

3.4.1 互斥算法

进程互斥的解决有两种做法, 一是由竞争各方平等协商; 二是引入进程管理者, 由管理者来协调竞争各方对互斥资源的使用。这两种做法在操作系统中都存在, 我们首先讨论第一种做法, 即基于进程间平等协商的互斥算法。

从多道系统开始, 程序的运行环境就存在资源共享问题。多个进程之间需要对有限的资源进行共享, 各进程在运行时是否能得到所需要的资源, 是受其他进程的影响的。如果一个进程所需要的资源已被其他进程占用, 该进程就无法正常运行下去。临界资源是指计算机系统上的需要互斥使用的硬件或软件资源, 如外设、共享代码段、共享数据结构等。多个进程在对临界资源进行访问时, 特别是进行写入或修改操作时, 必须互斥地进行。计算机系统中也有一些可以同时访问的共享资源不是临界资源, 如只读数据。

在多进程系统中, 我们可把进程间的相互制约关系按相互感知程度分成下面表3-2所列的三种类型。我们可以把计算机系统中资源共享的程度分成这样三个层次: 互斥(mutual exclusion)、死锁(deadlock)和饥饿(starvation)。保证资源的互斥使用是指多个进程不能同时使用同一个资源, 这是正确使用资源的最基本要求。避免死锁是指避免多个进程互不相让, 都得不到足够资源的情况出现, 从而保证系统功能的正常运行。避免饥饿是指避免某些进程一直得不到资源或得到资源的概率很小, 从而保障系统内资源使用的公平性。

表 3-2

相互感知的程度	交互关系	一个进程对其他进程的影响	潜在的控制问题
相互不感知(完全不了解其他进程的存在)	竞争(competition)	一个进程的操作对其他进程的结果无影响	互斥、死锁、饥饿
间接感知(双方都与第三方交互, 如共享资源)	通过共享进行协作	一个进程的结果依赖于从其他进程获得的信息	互斥、死锁、饥饿
直接感知(双方直接交互, 如通信)	通过通信进行协作	一个进程的结果依赖于从其他进程获得的信息	死锁、饥饿

为了保证临界资源的正确使用,我们可把临界资源的访问过程分成如图3-11所示的四个部分。

① 进入区(entry section):为了进入临界区使用临界资源,在进入区要检查可否进入临界区;如果可以进入临界区,通常设置相应的“正在访问临界区”标志,以阻止其他进程同时进入临界区。② 临界区(critical section):进程中访问临界资源的一段代码。③ 退出区(exit section):将“正在访问临界区”标志清除。④ 剩余区(remainder section):代码中的其余部分。

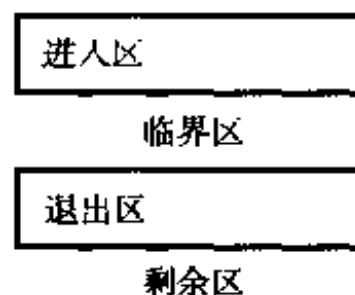


图3-11 临界区的访问过程

为了合理使用计算机系统资源,在操作系统中采用的进程同步机制应遵循以下几条准则。

① 空闲则入:任何同步机制都必须保证任何时刻最多只有一个进程位于临界区;当有进程位于临界区时,任何其他进程均不能进入临界区。② 忙则等待:当已有进程处于其临界区时,后到达的进程只能在进入区等待。③ 有限等待:为了避免死锁等现象的出现,等待进入临界区的进程不能无限期地“死等”。④ 让权等待:因在进入区等待而不能进入临界区的进程,应释放处理器,转换到阻塞状态,以使其他进程有机会得到处理器的使用权。

1. 进程互斥的软件方法

通过平等协商方式实现进程互斥的最初方法是软件方法。其基本思路是在进入区检查和设置一些标志,如果已有进程在临界区,则在进入区通过循环检查进行等待;在退出区修改标志。其中的主要问题是设置什么标志和如何检查标志。下面我们讨论几种用软件方法实现的软件互斥算法。

(1) 算法1:单标志算法

假设有两个进程 P_i 和 P_j 。设立一个公用整型变量 $turn$,描述允许进入临界区的进程标识。每个进程都在进入区循环检查变量 $turn$ 是否允许本进程进入。即 $turn$ 为 i 时,进程 P_i 可进入;否则循环检查该变量,直到 $turn$ 变为本进程标识。在退出区修改允许进入进程的标识。即进程 P_i 退出时,改 $turn$ 为进程 P_j 的标识 j 。图3-12所示为进程 P_i 的代码。

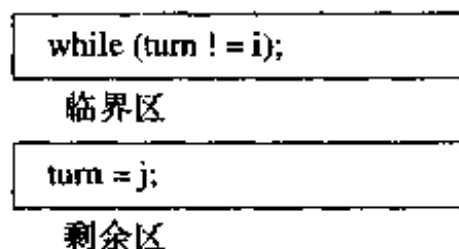


图3-12 互斥算法1(单标志)

算法1可以保证任何时刻最多只有一个进程在临界区。但它的缺点是强制轮流进入临界区,没有考虑进程的实际需要。这种算法容易造成资源利用不充分。例如,在 P_i 出让临界区之后, P_j 使用临界区之前, P_i 不可能再次使用临界区。

(2) 算法2:双标志、先检查算法

为了克服算法1的缺点,我们考虑修改临界区标志的设置。设立一个标志数组 $flag[]$,描述各进程是否在临界区,初值均为FALSE。在进入区的操作为:先检查,后修改。即在进入区先检查另一个进程是否在临界区,不在时修改本进程在临界区的标志,表示本进程在临界区。在退出区修改本进程在临界区的标志,表示本进程不在临界区。图3-13所示为进程 P_i 的代码。

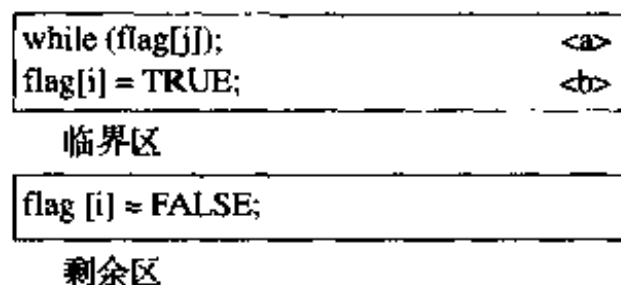


图3-13 互斥算法2(双标志、先检查)

算法2的优点是克服了算法1的缺点,两个进程不用交替进入,可连续使用。但由于使用多个

标志, 算法又产生一个新问题, 即进程 P_i 和 P_j 可能同时进入临界区, 从而违反了最多只有一个进程在临界区的要求。我们按序列“ $P_i\langle a \rangle P_j\langle a \rangle P_i\langle b \rangle P_j\langle b \rangle$ ”执行时, 就会出现进程 P_i 和 P_j 同时进入的问题。即进程在检查对方标志flag之后和切换自己标志flag之前有一段时间间隔, 这个时间间隔导致两个进程都在进入区通过检查。这个问题出在检查和修改操作不能连续进行。

(3) 算法3: 双标志、先修改后检查算法

为了解决算法2的新问题; 我们有两种选择: 一是保证检查和修改操作之间不出现间隔, 二是修改标志含义。第一种方法是仅使用软件方法无法做到的, 我们采用第二种方法。算法3类似于算法2, 它们的区别在于进入区操作是先修改后检查。这时标志flag[i]表示进程 P_i 想进入临界区, 而不再表示进程i在临界区。

图3-14所示为进程 P_i 的代码

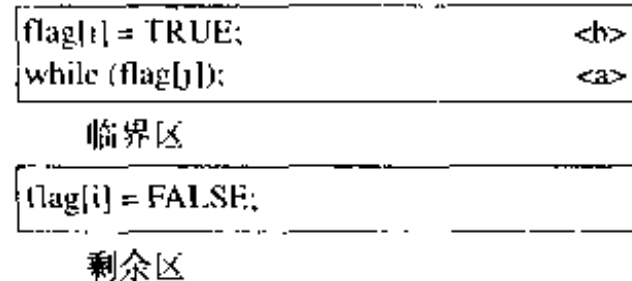


图3-14 互斥算法3（双标志、先修改后检查）

算法3可防止两个进程同时进入临界区。但它的缺点是 P_i 和 P_j 可能都进入不了临界区。我们按序列“ $P_i\langle b \rangle P_j\langle b \rangle P_i\langle a \rangle P_j\langle a \rangle$ ”执行时, 就会都进不了临界区。即在修改本进程标志flag之后和检查对方flag之前有一段时间间隔, 这个间隔导致两个进程都想进入临界区, 从而在检查对方标志时不通过。

(4) 算法4: 先修改、后检查、后修改者等待算法

算法4的基本思想是结合算法1和算法3。标志flag[i]表示进程 P_i 想进入临界区, 标志turn表示同时修改标志时要在进入区等待的进程标识。在进入区先修改后检查, 通过修改同一标志turn来描述标志修改的先后; 检查对方标志flag, 如果对方不想进入临界区则自己进入; 否则再检查标志turn, 由于标志turn中保存的是较晚的一次赋值, 因此较晚修改标志的进程等待, 较早修改标志的进程进入临界区。图3-15所示为进程 P_i 的代码。

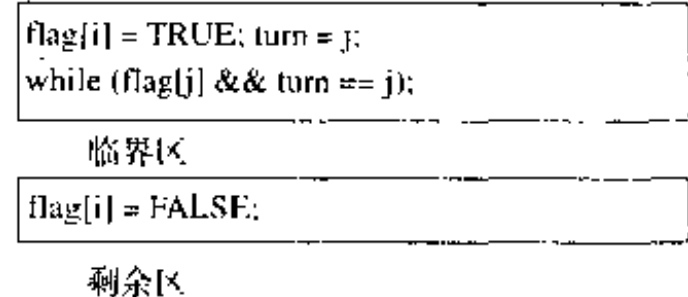


图3-15 互斥算法4（先修改、后检查、后修改者等待）

至此, 算法4可完全正常工作。即实现了同步机制要求的四条准则中的前两条: 空闲则入和忙则等待。但从上面的软件实现方法中可发现, 对于两个进程间的互斥和三个以上进程间的互斥的进入区是要区别对待的, 而这里最主要的问题就是修改标志和检查标志不能作为一个整体不被执行。下面的硬件方法就是利用处理器的指令系统来解决这个问题。

2. 进程互斥的硬件方法

完全利用软件方法实现进程互斥有很大局限性, 如不适用于数目很多的进程间的互斥。现在已很少单独采用软件方法。在平等协商时都利用某些硬件指令来实现进程互斥。硬件方法的主要思路是用一条指令完成读和写两个操作, 因而保证读操作与写操作不被打断。依据所采用的指令的不同硬件方法分成TS指令和Swap指令两种。

(1) TS (Test-and-Set)指令

TS指令的功能是读出指定标志后把该标志设置为TRUE。TS指令的功能可描述成下面的函数。

```

boolean TS(boolean *lock) {
    boolean old;
    old = *lock; *lock = TRUE;
    return old;
}

```

利用TS指令实现的进程互斥算法是，每个临界资源设置一个公共布尔变量lock，表示资源的两种状态：TRUE表示正被占用，FALSE表示空闲，初值为FALSE。在进入区利用TS进行检查和修改标志lock；有进程在临界区时，重复检查，直到其他进程退出时，检查通过。如图3-16所示，所有要访问临界资源的进程的进入区和退出区代码都是相同的。

(2) Swap指令（或Exchange指令）

Swap指令的功能是交换两个字（字节）的内容。我们可用下面的函数描述Swap指令的功能。

```

void SWAP(int *a, int *b) {
    int temp;
    temp = *a; *a = *b; *b = temp;
}

```

利用Swap指令实现的进程互斥算法是，每个临界资源设置一个公共布尔变量lock，初值为FALSE；每个进程设置一个私有布尔变量key，用于与lock间的信息交换。在进入区利用Swap指令交换lock与key的内容，然后检查key的状态；有进程在临界区时，重复交换和检查过程，直到其他进程退出时，检查通过。图3-17显示的是所有要访问临界资源的进程的相关代码。

与前面的软件方法相比，硬件方法由于采用处理器指令很好地把修改和检查操作结合成一个不可分的整体而具有明显的优点。具体而言，硬件方法的优点体现在以下几个方面：① 适用范围广：硬件方法适用于任意数目的进程，在单处理器和多处理器环境中完全相同；② 简单：硬件方法的标志设置简单，含义明确，容易验证其正确性；③ 支持多个临界区：在一个进程内有多个临界区时，只需为每个临界区设立一个布尔变量。

硬件方法有许多优点，但也有一些自身无法克服的缺点。这些缺点主要包括：① 进程在等待进入临界区时要耗费处理器时间，不能实现“让权等待”；② 由于进入临界区的进程是从等待进程中随机选择的，有的进程可能一直选不上，从而导致“饥饿”。

3.4.2 信号量

前面的互斥算法都是平等进程间的协商机制，它们存在的问题是平等协商无法解决的，需要引入一个地位高于进程的管理者来解决公有资源的使用问题。操作系统可以从进程管理者的角度来处理互斥的问题，信号量(semaphore)就是由操作系统提供的管理公有资源的有效手段。信号量代表可用资源实体的数量。

```

while TS (&lock);
    临界区
lock = FALSE;
    剩余区

```

图3-16 互斥算法（TS指令）

```

key = TRUE;
do
{
    SWAP (&lock, &key);
} while (key);
    临界区
lock = FALSE;
    剩余区

```

图3-17 互斥算法（Swap指令）

1. 信号量和P、V原语

信号量是荷兰学者Dijkstra于1965年提出的一种卓有成效的进程同步机制。信号量机制所使用的P、V原语就来自荷兰语的test(proberen)和increment(verhogen)。每个信号量s除一个整数值s.count(计数)外,还有一个进程等待队列s.queue,其中是阻塞在该信号量的各个进程的标识。信号量只能通过初始化和两个标准的原语来访问。作为操作系统核心代码的一部分,P、V原语的执行,不受进程调度和执行的打断,从而很好地解决了原语操作的整体性的问题。信号量的初始化可指定一个非负整数值,表示空闲资源总数。若信号量为非负整数值,该值表示当前的空闲资源数;若为负值,其绝对值表示当前等待临界区的进程数。

依据我们对临界区访问过程的分析,信号量机制中P原语相当于进入区操作,V原语相当于退出区操作。下面我们分析操作系统对这两个原语操作的处理过程。

P原语所执行的操作可用下面函数wait(s)来描述。

```
wait(s)
{
    --s.count;           //表示申请一个资源;
    if (s.count < 0) //表示没有空闲资源;
    {
        调用进程进入等待队列s.queue;
        阻塞调用进程;
    }
}
```

V原语所执行的操作可用下面函数signal(s)来描述。

```
signal(s)
{
    ++s.count;           //表示释放一个资源;
    if (s.count <= 0)    //表示有进程处于阻塞状态;
    {
        从等待队列s.queue中取出头一个进程P;
        进程P进入就绪队列;
    }
}
```

利用操作系统提供的信号量机制,可实现临界资源的互斥访问。如图3-18所示,我们为临界资源设置一个互斥信号量mutex(MUTual Exclusion),其初值为1;在每个进程中将临界区代码置于P(mutex)和V(mutex)原语之间。

在使用信号量进行共享资源访问控制时,必须成对使用P、V原语。遗漏P原语则不能保证互斥访问,遗漏V原语则不能在使用临界资源之后将其释放给其他等待的进程。P、V原语的使用不能次序错误、重复或遗漏。

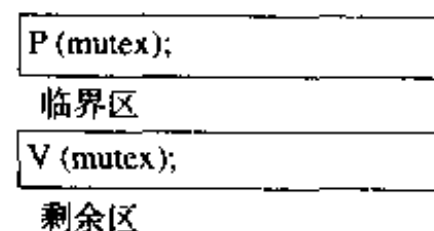


图3-18 互斥算法(信号量)

利用操作系统提供的信号量机制,可实现进程间的同步,即所谓的前趋关系。如图3-19所示,

前趋关系是指并发执行的进程 P_1 和 P_2 中, 分别有代码 C_1 和 C_2 , 要求 C_1 在 C_2 开始前完成执行。我们可为每个前趋关系设置一个互斥信号量 S_{12} , 其初值为0。这样, 只有在 P_1 执行到 $V(S_{12})$ 后, P_2 才会结束 $P(S_{12})$ 的执行。

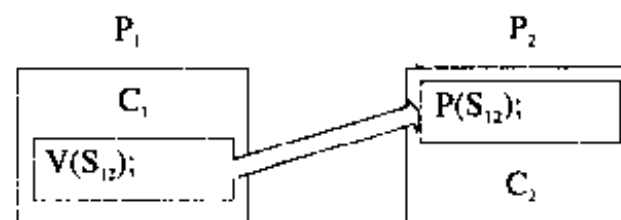


图3-19 利用信号量描述前趋关系

2. 信号量集

当利用信号量机制解决了单个资源的互斥访问后, 我们讨论如何控制同时需要多个资源的互斥访问。信号量集是指同时需要多个资源时的信号量操作。

(1) AND型信号量集

AND型信号量集是指同时需要多个资源且每种占用一个资源时的信号量操作。当一段处理代码需要同时获取两个或多个临界资源时, 就可能出现由于各进程分别获得部分临界资源并等待其余的临界资源的局面。各进程都会“各不相让”, 从而出现死锁。解决这个问题的一個基本思路是: 在一个原语中申请整段代码需要的多个临界资源, 要么全部分配给它, 要么一个都不分配给它。这就是AND型信号量集的基本思想。我们称AND型信号量集P原语为Swait (Simultaneous Wait), V原语为Ssignal (Simultaneous Signal)。在Swait时, 各个信号量的次序并不重要, 虽然会影响进程归入哪个阻塞队列, 但是因为是对资源全部分配或不分配, 所以总有进程获得全部资源并在推进之后释放资源, 因此不会死锁。下面是Swait和Ssignal的伪代码。

```
Swait(S1, S2, ..., Sn) //P原语;
{
while (TRUE)
{
if (S1 >= 1 && S2 >= 1 && ... && Sn >= 1)
{
//满足资源要求时的处理;
for (i = 1; i <= n; ++i) --Si;
//注: 与wait的处理不同, 这里是在确信可满足资源要求时, 才进行减1操作;
break;
}
}
else
{
//某些资源不够时的处理;
调用进程进入第一个小于1信号量的等待队列Si.queue;
阻塞调用进程;
}
}
}

Ssignal(S1, S2, ..., Sn) //V原语;
{
for (i = 1; i <= n; ++i)
{
++Si; //释放占用的资源;
for (each process P waiting in Si.queue)
```

```

        //检查每种资源的等待队列的所有进程;
    }
    从等待队列 $S_i.queue$ 中取出进程 $P$ ;
    if (判断进程 $P$ 是否通过 $Swait$ 中的测试)
        //注:与signal不同,这里要进行重新判断;
        //通过检查(资源够用)时的处理;
        进程 $P$ 进入就绪队列;
    }
    else
        ( //未通过检查(资源不够用)时的处理;
        进程 $P$ 进入某等待队列;
        )
    }
}
}

```

(2) 一般“信号量集”

一般“信号量集”是指同时需要多种资源、每种占用的数目不同、且可分配的资源还存在一个临界值时的信号量处理。由于一次需要 n 个某类临界资源,因此如果通过 n 次wait操作申请这 n 个临界资源,操作效率很低,并可能出现死锁。一般信号量集的基本思路就是在AND型信号量集的基础上进行扩充,在一次原语操作中完成所有的资源申请。进程对信号量 S_i 的测试值为 t_i (表示信号量的判断条件,要求 $S_i \geq t_i$;即当资源数量低于 t_i 时,便不予分配),占用值为 d_i (表示资源的申请量,即 $S_i = S_i - d_i$)。对应的P、V原语格式为:

```

Swait( $S_1, t_1, d_1; \dots; S_n, t_n, d_n$ );
Ssignal( $S_1, d_1; \dots; S_n, d_n$ );

```

一般“信号量集”可以用于各种情况的资源分配和释放。下面是几种特殊的情况:

1) Swait(S, d, d)表示每次申请 d 个资源,当资源数量少于 d 个时,便不予分配。

2) Swait($S, 1, 1$)表示互斥信号量。

3) Swait($S, 1, 0$)可作为一个可控开关(当 $S \geq 1$ 时,允许多个进程进入临界区;当 $S=0$ 时,禁止任何进程进入临界区)。

由于一般信号量在使用时的灵活性,因此通常并不成对使用Swait和Ssignal。为了避免死锁,可一起申请所有需要的资源,但不一起释放。

3.4.3 经典进程同步问题

本节我们讨论几个利用信号量来实现进程互斥和同步的经典例子。这里的主要问题是如何选择信号量和如何安排P、V原语的使用顺序。

依据信号量与进程的关系,我们可把进程中使用的信号量分成私有信号量和公用信号量。私有信号量是指只与制约进程和被制约进程有关的信号量;公用信号量是指与一组并发进程有关的信号量。

1. 生产者—消费者问题

生产者—消费者问题(producer-consumer problem)是指若干进程通过有限的共享缓冲区交换数据时的缓冲区资源使用问题。假设“生产者”进程不断向共享缓冲区写入数据(即生产数据),而“消费者”进程不断从共享缓冲区读出数据(即消费数据);共享缓冲区共有n个;任何时刻只能有一个进程可对共享缓冲区进行操作。所有生产者和消费者之间要协调,以完成对共享缓冲区的操作。如图3-20所示。

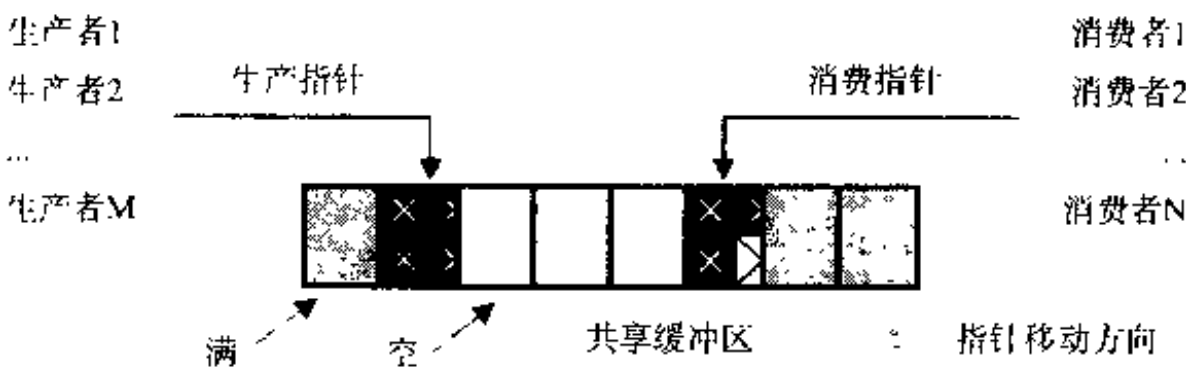


图3-20 生产者—消费者问题

我们可把共享缓冲区中的n个缓冲块视为共享资源,生产者写入数据的缓冲块成为消费者的可用资源,而消费者读出数据后的缓冲块成为生产者的可用资源。为此,可设置三个信号量:full、empty和mutex。其中:full表示有数据的缓冲块数目,初值是0;empty表示空的缓冲块数目,初值是n;mutex用于访问缓冲区时的互斥,初值是1。实际上,full和empty间存在如下关系:full + empty == n。图3-21所示为生产者与消费者的算法。

生产者	消费者
P(empty);	P(full);
P(mutex); //进入区	P(mutex); //进入区
one unit -> buffer;	one unit <-- buffer;
V(mutex);	V(mutex);
V(full); //退出区	V(empty); //退出区

图3-21 利用信号量的生产者与消费者算法

注意:这里每个进程中各个P操作的次序是重要的、各进程必须先检查自己对应的资源数目,在确信有可用资源后再申请对整个缓冲区的互斥操作;否则,先申请对整个缓冲区的互斥操作,后申请自己对应的缓冲块资源,就可能死锁。出现死锁的条件是,申请到对整个缓冲区的互斥操作后,才发现自己对应的缓冲块资源,这时已不可能放弃对整个缓冲区的占用。

如果采用AND信号量集,相应的进入区和退出区都很简单。如生产者的进入区为Swait(empty, mutex),退出区为Ssignal(full, mutex)。

2. 读者—写者问题

读者—写者问题(readers-writers problem)是指多个进程对一个共享资源进行读写操作的问题。假设“读者”进程可对共享资源进行读操作,“写者”进程可对共享资源进行写操作;任一时刻“写者”最多只允许一个,而“读者”则允许多个。即对共享资源的读写操作限制关系包括:“读

“写—写”互斥、“写—读”互斥和“读—读”允许。

我们可认为写者之间、写者与第一个读者之间要对共享资源进行互斥访问，而后续读者不需要互斥访问。为此，可设置两个信号量Wmutex、Rmutex和一个公共变量Rcount。其中：Wmutex表示“允许写”，初值是1；公共变量Rcount表示“正在读”的进程数，初值是0；Rmutex表示对Rcount的互斥操作，初值是1。图3-22所示为读者—写者算法。

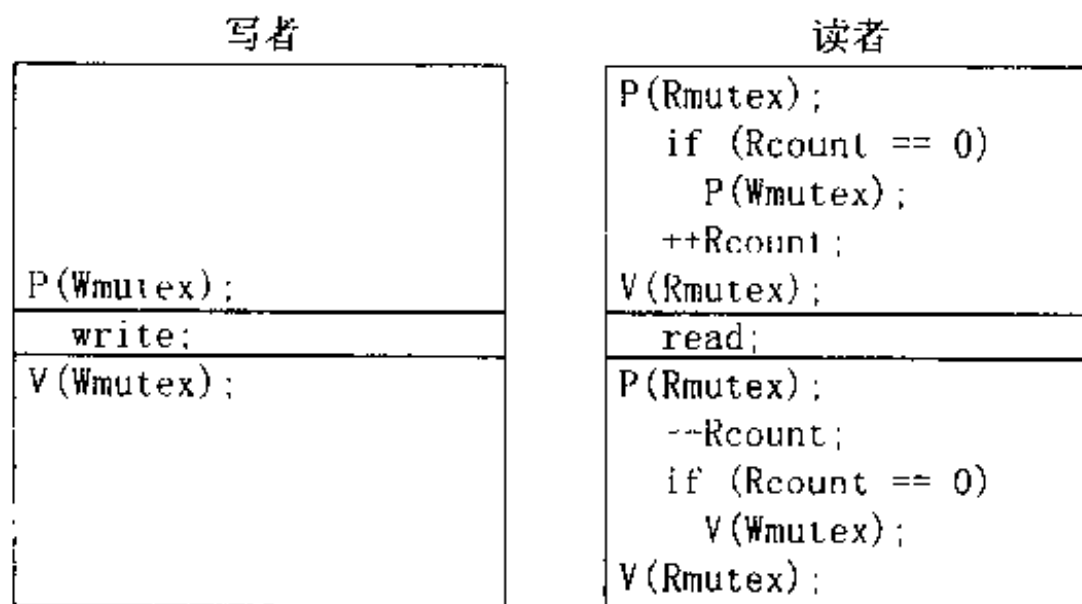


图3-22 读者—写者算法（信号量）

在这个例子中，我们可见到临界资源访问过程的嵌套使用。在读者算法中，进入区和退出区又分别嵌套了一个临界资源访问过程。

对读者—写者问题，也可采用一般“信号量集”机制来实现。如果我们在前面的读写操作限制上再加一个限制条件：同时读的“读者”最多R个。这时，可设置两个信号量Wmutex和Rcount。其中：Wmutex表示“允许写”，初值是1；Rcount表示“允许读者数目”，初值为R。图3-23所示为采用一般“信号量集”机制来实现的读者—写者算法。

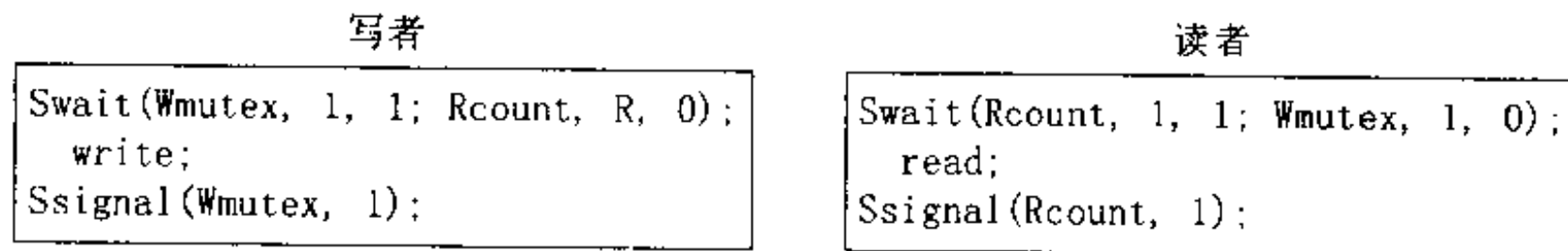


图3-23 读者—写者算法（信号量集）

3.4.4 管程

用信号量可实现进程间的同步和互斥，但由于信号量的控制分布在整个程序中，其正确性分析很困难。管程就是为了解决信号量机制面临的困难而提出的一种新的进程间同步机制，它把对信号量的控制集中在管程内部，保证进程互斥地访问共享变量，并方便地阻塞和唤醒进程。相比之下，管程比信号量好控制。

首先，我们来分析一下信号量机制存在的问题。从前面的例子可以观察到，信号量机制的同

步操作是分散在各进程中的，使用不当就可能导致各进程死锁（如P、V操作的次序错误、重复或遗漏）。使用信号量机制的程序代码易读性差，要了解对一组共享变量及信号量的操作是否正确，必须通读整个系统或者并发执行的多个程序。信号量机制的使用不利于代码的修改和维护，各模块的独立性差，任一组变量或一段代码的修改都可能影响全局。信号量机制的正确性难以保证，操作系统或并发程序通常会使用很多信号量，它们的关系错综复杂，很难保证这样一个复杂的系统没有逻辑错误。

为了解决信号量机制的这些问题，Hoare和Hanson于1973年提出了管程机制。管程的基本思想是把信号量及其操作原语封装在一个对象内部。即，将共享资源以及针对共享资源能够进行的所有操作集中在一个模块中。可把“管程”(monitor)定义为关于共享资源的数据结构以及一组针对该资源的操作过程所构成的软件模块。管程可以以函数库的形式实现，一个管程就是一个基本程序单位，可以单独编译。管程中引入了面向对象的思想，一个管程不仅有关于共享资源的数据结构，而且还有对数据结构进行操作的代码。管程对共享资源进行了封装，进程可调用管程中定义的操作过程，而这些操作过程的实现，在管程外部是不可见的。

引入管程可得到多方面的好处。首先，由于管程的良好封装特征，因此可增强模块的独立性。按资源管理的观点，可把系统分解成若干模块，用数据表示抽象系统资源；同时利用共享资源和专用资源在管理上的差别，按不同的管理方式定义模块的类型和结构，使同步操作相对集中，从而增加了模块的相对独立性。其次，引入管程可提高代码的可读性，便于修改和维护，正确性易于保证。由于采用集中式同步机制，一个操作系统或并发程序由若干个这样的模块所构成，一个模块通常较短，模块之间关系清晰，有利于正确性的保证。

在管程的实现中，为了保证管程共享变量的数据完整性，需要保证管程的互斥进入，并在管程中设置进程等待队列以及相应的等待及唤醒操作，进程只能通过调用管程中所说明的外部过程来间接地访问管程中的共享变量。当一个进入管程的进程执行等待操作时，它应当释放管程的互斥权；当一个进入管程的进程执行唤醒操作时（如P唤醒Q），管程中就存在两个同时处于活动状态的进程。为此，每个管程都设置有一个入口等待队列和一个紧急等待队列。当一个进程试图进入一个已被占用的管程时，它在管程的入口处等待，这个在管程的入口处的进程等待队列称作入口等待队列。在管程内部，由于执行唤醒操作，因此可能会出现多个等待进程，它们已被唤醒，但由于管程的互斥进入而等待，这个等待队列称为紧急等待队列。紧急等待队列的优先级高于入口等待队列的优先级。

在此，我们需要说明管程与进程的区别。在操作系统中设置进程是为了描述程序的动态执行过程，而设置管程是为了进行进程的同步，协调进程的相互关系和对共享资源进行访问。操作系统维护的进程数据结构是进程控制块，而与管程相关的数据结构是等待队列。管程可被进程调用。管程与操作系统中的共享资源相关，没有创建和撤消；而进程有创建和撤消。

3.4.5 Windows 2000/XP的进程互斥和同步

在Windows 2000/XP中提供了互斥对象、信号量对象和事件对象三种同步对象和相应的系统调用，用于进程和线程的同步。这些同步对象都有一个用户指定的对象名称，不同进程中用同样

的对象名称来创建或打开对象，从而获得该对象在本进程的句柄。从本质上讲，这组同步对象的功能是相同的，它们的区别在于适用场合和效率会有所不同。

互斥对象(Mutex)就是互斥信号量，在一个时刻只能被一个线程使用。它的相关API包括：CreateMutex、OpenMutex和ReleaseMutex。CreateMutex创建一个互斥对象，返回对象句柄；OpenMutex打开并返回一个已存在的互斥对象句柄，用于后续访问；而ReleaseMutex释放对互斥对象的占用，使之成为可用。

信号量对象(Semaphore)就是资源信号量，初始值的取值在0到指定最大值之间，用于限制并发访问的线程数。它的相关API包括：CreateSemaphore、OpenSemaphore和ReleaseSemaphore。CreateSemaphore创建一个信号量对象，在输入参数中指定最大值和初值，返回对象句柄；OpenSemaphore返回一个已存在的信号量对象句柄，用于后续访问；ReleaseSemaphore释放对信号量对象的占用。

事件对象(Event)相当于“触发器”，可用于通知一个或多个线程某事件的出现。它的相关API包括：CreateEvent、OpenEvent、SetEvent、ResetEvent和PulseEvent。CreateEvent创建一个事件对象，返回对象句柄；OpenEvent返回一个已存在的事件对象句柄，用于后续访问；SetEvent和PulseEvent设置指定事件对象为可用状态；ResetEvent设置指定事件对象为不可用状态。

对于这三种同步对象，Windows 2000/XP提供了两个统一的等待操作WaitForSingleObject和WaitForMultipleObjects。WaitForSingleObject可在指定的时间内等待指定对象为可用状态；WaitForMultipleObjects可在指定的时间内等待多个对象为可用状态。这两个API的接口为：

```
DWORD WaitForSingleObject( HANDLE hHandle, // 等待对象句柄;
    DWORD dwMilliseconds // 以毫秒为单位的最长等待时间;
);

DWORD WaitForMultipleObjects( DWORD nCount,
    // 对象句柄数组中的句柄数;
    CONST HANDLE *lpHandles,
    // 指向对象句柄数组的指针，数组中可包括多种对象句柄;
    BOOL bWaitAll,
    // 等待标志：TRUE表示所有对象同时可用，FALSE表示至少一个对象可用;
    DWORD dwMilliseconds // 等待超时时限;
);
```

除了上述三种同步对象，Windows 2000/XP还提供了一些与进程同步相关的机制，如临界区对象和互锁变量访问API等。临界区(Critical Section)对象只能用于在同一进程内使用的临界区，同一进程内各线程对它的访问是互斥进行的。把变量说明为CRITICAL_SECTION类型，就可作为临界区使用。相关API包括：InitializeCriticalSection、EnterCriticalSection、TryEnterCriticalSection、LeaveCriticalSection和DeleteCriticalSection。InitializeCriticalSection对临界区对象进行初始化；EnterCriticalSection等待占用临界区的使用权，得到使用权时返回；TryEnterCriticalSection非等待方式申请临界区的使用权，申请失败时返回0；LeaveCriticalSection释放临界区的使用权；DeleteCriticalSection释放与临界区对象相关的所有系统资源。

互锁变量访问API相当于硬件指令，用于对整型变量的操作，可避免线程间切换对操作连续性的影响。这组互锁变量访问API包括：InterlockedExchange、InterlockedCompareExchange、InterlockedExchangeAdd、InterlockedDecrement、InterlockedIncrement。InterlockedExchange进行32位数据的先读后写原子操作；InterlockedCompareExchange依据比较结果进行赋值的原子操作；InterlockedExchangeAdd先加后存结果的原子操作；InterlockedDecrement先减1后存结果的原子操作；InterlockedIncrement先加1后存结果的原子操作。

3.5 进程间通信

进程间通信(Inter-Process Communication, IPC)要解决的问题是进程间的信息交流。这种信息交流的量可大可小。操作系统提供了多种进程间通信机制，可分别适用于多种不同的场合。前面所介绍的进程同步机制实际上就是进程间通信的一种，只不过交流的信息量非常少。

按通信量的大小，我们可把进程间通信分成低级通信和高级通信。在低级通信中，进程间只能传递状态和整数值（控制信息），包括进程互斥和同步所采用的信号量机制。它的优点是速度快，缺点是传送信息量小、通信效率低、编程复杂。由于每次通信传递的信息量固定，因此如果传递较多信息则需要多次通信。用户直接实现通信的细节，编程复杂，容易出错。在高级通信中，进程间可传送任意数量的数据，包括共享存储区、管道和消息等机制。

按通信过程中是否有第三方作为中转，我们可把进程间通信分成直接通信和间接通信。直接通信是指发送方把信息直接传递给接收方。在直接通信方式中，发送方要指定接收方的地址或标识，也可以指定多个接收方或广播式地址；接收方可接收来自任意发送方的消息，并在读出消息的同时获取发送方的地址。间接通信是指通信过程要借助于收发双方进程之外的共享数据结构（如消息队列）作为通信中转。在间接通信方式中，接收方和发送方的数目可以是任意的。

进程间通信还要考虑到通信过程中的一些其他特征，如通信链路特征、数据格式和收发双方的同步方式等。对进程间通信模式有影响的通信链路特征包括：链路是点对点还是广播链路；通信链路是否带缓冲区；链路是单向还是双向等。数据格式主要分成字节流和报文两类。采用字节流格式时，接收方不保留各次发送之间的分界；采用报文格式时，接收方保留各次发送之间的分界。报文方式还可进一步分成定长报文/不定长报文和可靠报文/不可靠报文。收发操作的同步方式可分成阻塞和不阻塞两种。阻塞操作是指操作方要等待操作结束；不阻塞操作是指操作提交后立即返回。

3.5.1 Windows 2000/XP的信号

信号(signal)是进程与外界的一种低级通信方式，相当于进程的“软件”中断。进程可发送信号，每个进程都有指定信号处理例程。信号通信是单向的和异步的。Windows 2000/XP有两组与信号相关的系统调用，分别处理不同的信号。

1. SetConsoleCtrlHandler和GenerateConsoleCtrlEvent

SetConsoleCtrlHandler可定义或取消本进程的信号处理例程(HandlerRoutine)列表中的用户定义例程。例如，缺省时，每个进程都有一个信号CTRL+C的处理例程，我们可利用SetConsoleCtrl

Handler调用来忽视或恢复对CTRL+C的处理。GenerateConsoleCtrlEvent可发送信号到与本进程共享同一控制台的控制台进程组。这一组系统调用处理的信号包括表3-3中的5种信号。

表 3-3

信号名	说明
CTRL_C_EVENT	收到CTRL+C信号
CTRL_BREAK_EVENT	收到CTRL+BREAK信号
CTRL_CLOSE_EVENT	当用户关闭控制台时系统向该控制台的所有进程发送的控制台关闭信号
CTRL_LOGOFF_EVENT	用户退出系统时系统向所有控制台进程发送的退出信号
CTRL_SHUTDOWN_EVENT	系统关闭时系统向所有控制台进程发送的关机信号

2. signal和raise

signal用于设置中断信号处理例程。raise用于发送信号。这一组系统调用处理的信号包括表3-4中列出的6种信号。这6种信号是与传统的UNIX系统相同的，而前面一组系统调用处理的5种信号是Windows 2000/XP中特有的。

表 3-4

信号名	说明
SIGABRT	非正常终止
SIGFPE	浮点计算错误
SIGILL	非法指令
SIGINT	CTRL+C信号(对Win32无效)
SIGSEGV	非法存储访问
SIGTERM	终止请求

3.5.2 Windows 2000/XP基于文件映射的共享存储区

共享存储区(shared memory)可用于进程间的大数据量通信。进行通信的各进程可以任意读写共享存储区，也可在共享存储区上使用任意数据结构。在使用共享存储区时，需要进程互斥和同步机制的辅助来确保数据一致性。Windows 2000/XP采用文件映射(file mapping)机制来实现共享存储区，用户进程可以将整个文件映射为进程虚拟地址空间的一部分来加以访问。

下面系统调用与共享存储区的使用相关。CreateFileMapping为指定文件创建一个文件映射对象，返回对象指针；OpenFileMapping打开一个命名的文件映射对象，返回对象指针；MapViewOfFile把文件映射到本进程的地址空间，返回映射地址空间的首地址；FlushViewOfFile可把映射地址空间的内容写到物理文件中；UnmapViewOfFile拆除文件与本进程地址空间之间的映射关系；CloseHandle可关闭文件映射对象。当完成文件到进程地址空间的映射后，就可利用首地址进行读写。在信号量等机制的辅助下，通过一个进程向共享存储区写入数据而另一个进程从共享存储区读出数据，就可在两个进程间实现大量数据的交流。

3.5.3 Windows 2000/XP管道

管道(pipe)是一条在进程间以字节流方式传送的通信通道。它是利用操作系统核心的缓冲区(通常几十个KB)来实现的一种单向通信,常用于命令行所指定的输入输出重定向和管道命令。在使用管道前要建立相应的管道,然后才可使用。

Windows 2000/XP提供无名管道和命名管道两种管道机制。Windows 2000/XP的无名管道类似于UNIX系统的管道,但提供的安全机制比UNIX管道完善。利用CreatePipe可创建无名管道,并得到两个读写句柄;然后利用ReadFile和WriteFile可进行无名管道的读写。下面是CreatePipe的调用格式。

```
BOOL CreatePipe( PHANDLE hReadPipe, // 读句柄;
                PHANDLE hWritePipe, // 写句柄;
                LPSECURITY_ATTRIBUTES lpPipeAttributes, // 安全属性指针;
                DWORD nSize          // 管道缓冲区字节数;
                );
```

Windows 2000/XP的命名管道是服务器进程与一个客户进程间的一条通信通道,可实现不同机器上的进程通信。它采用客户-服务器模式连接本机或网络中的两个进程。在建立命名管道时,存在一定的限制。即服务器方(创建命名管道的一方)只能在本机上创建命名管道,命名方式只能是“\\.\pipe\PipeName”形式,不能在其他机器上创建管道;但客户方(连接到一个命名管道实例的一方)可以连接到其他机器上的命名管道,命名方式可为“\\serverName\pipe\pipename”形式。服务器进程为每个管道实例建立单独的线程或进程。下面是与命名管道相关的主要系统调用。CreateNamedPipe在服务器端创建并返回一个命名管道句柄;ConnectNamedPipe在服务器端等待客户进程的请求;CallNamedPipe从管道客户进程建立与服务器的管道连接;ReadFile、WriteFile(用于阻塞方式)、ReadFileEx、WriteFileEx(用于非阻塞方式)用于命名管道的读写。

3.5.4 Windows 2000/XP邮件槽

Windows 2000/XP中提供的邮件槽(mailslot)是一种不定长、不可靠的单向消息通信机制。消息的发送不需要接收方准备好,随时可发送。邮件槽也采用客户-服务器模式,只能从客户进程发往服务器进程。服务器进程负责创建邮件槽,它可从邮件槽中读消息;而客户进程可利用邮件槽的名字向它发送消息。在建立邮件槽时,也存在一定的限制。即服务器进程(接收方)只能在本机建立邮件槽,命名方式只能是“\\.\mailslot\[path]name”方式;但客户进程(发送方)可打开其他机器上的邮件槽,命名方式可为“\\range\mailslot\[path]name”,这里range可以是本机、其他机器的名字或域名。下面是与邮件槽相关的主要系统调用。CreateMailslot服务器方创建邮件槽,返回其句柄;GetMailslotInfo服务器查询邮件槽的信息,如消息长度、消息数目、读操作等待时限等;SetMailslotInfo服务器设置读操作等待时限;ReadFile服务器读邮件槽;CreateFile客户方打开邮件槽;WriteFile客户方发送消息。由于邮件槽不提供可靠的传输机制,因此在邮件槽

关闭过程中可能出现信息丢失的情况。在邮件槽的所有服务器句柄关闭后，邮件槽被关闭，如果这时还有未读出的消息，这些消息将会被丢弃，所有客户句柄都被关闭。

3.5.5 套接字

套接字(socket)是一种网络通信机制，它通过网络在不同计算机上的进程间进行双向通信。套接字所采用的数据格式可为可靠的字节流或不可靠的报文，通信模式可为客户/服务器模式或对等模式。为了实现不同操作系统上的进程通信，需要约定网络通信时不同层次的通信过程和信息格式，TCP/IP协议就是广泛使用的网络通信协议。

在UNIX系统中使用的BSD套接字主要是基于TCP/IP协议，操作系统中有一组标准的系统调用完成通信连接的维护和数据收发。例如，send和sendto用于数据发送，而recv和recvfrom用于数据接收。

Windows 2000/XP中的套接字规范称为“Winsock”，它除了支持标准的BSD套接字外，还实现了一个真正与协议独立的应用程序编程接口，可支持多种网络通信协议。例如，在WinSock 2.2中分别把send、sendto、recv和recvfrom扩展成WSASend、WSASendto、WSARecv和WSARecvfrom。

3.6 死锁问题

死锁(deadlock)是指系统中多个进程无限制地等待永远不会发生的条件。在这一节中，我们将讨论如何在操作系统中处理死锁问题。

3.6.1 概述

产生死锁的根本原因是对互斥资源的共享，并发执行进程的同步关系不当。为了仔细分析死锁的形成过程，我们把进程使用的资源分成可重用资源和可消耗资源两类。对于可重用资源(reusable resource)，每个时刻只有一个进程使用，在宏观上各个进程轮流使用。如处理器、主存和辅存、I/O通道、外设、数据结构（如文件、数据库和信号量等）都是可重用资源。在可重用资源使用时出现的死锁，都是由于各进程都拥有部分资源，同时在请求其他进程已占有的资源，从而造成永远等待。例如，假设系统中的资源A和B都只有一个，则如图3-24所示的申请次序“P1<a> P2<a> P1 P2”就会形成死锁。

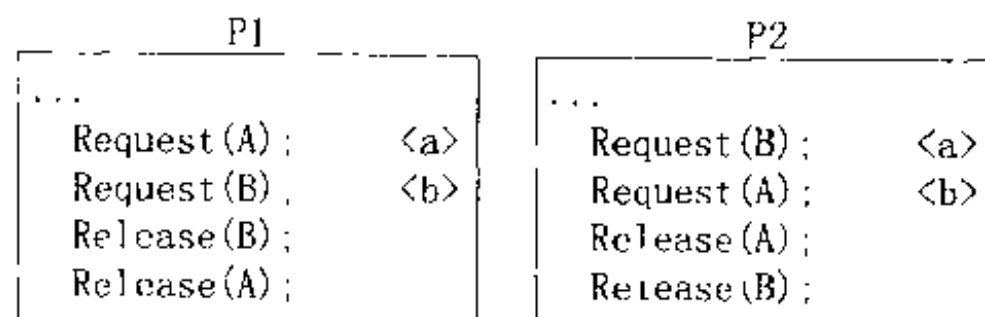


图3-24 可重用资源死锁

可消耗资源(consumable resource)是指可以动态生成和消耗资源，一般不限制数量。如硬件

中断、信号、消息、缓冲区内的数据等都是可消耗资源。由于可消耗资源的生成和消耗存在依赖关系，因此它们的使用也可能因为双方都等待对方生成资源而形成死锁。如图3-25所示的执行次序“P1<a> P2<a>”就会由于对方还未生成资源而形成永远等待。

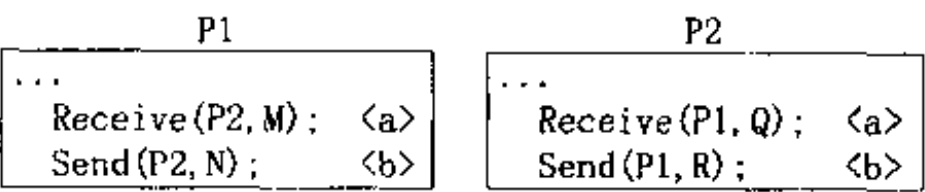


图3-25 可消耗资源死锁

由上而的分析可知，当多个进程并发执行时，由于相互的推进顺序的不确定，很可能会导致死锁。为此，我们需要讨论死锁的形成原因和可能的解决方案。

我们可把死锁发生条件描述成下列四个条件。它们是产生死锁的充分必要条件，只有四个条件都满足时，才会出现死锁。

- 1) 互斥：任一时刻只允许一个进程使用资源。
- 2) 请求和保持：进程在请求其余资源时，不主动释放已经占用的资源。
- 3) 非剥夺：进程已经占用的资源，不会被强制剥夺。
- 4) 环路等待：环路中的每一条边是进程在请求另一进程已经占有的资源。

在分析了产生死锁的四个条件后，我们可使用表3-5中的三种基本方法来处理死锁。下而我们将具体讨论如何使用这三种方法来处理死锁问题。

表 3-5

方 法	资源分配策略	各种可能模式	主要 优点	主要 缺点
预防(Prevention)	保守的；宁可资源 闲置（从机制上使死 锁条件不成立）	一次请求所有资 源<条件2>	适用于作为突发 式处理的进程；不 必剥夺	效率低；进程初始化 时间延长
		资源剥夺<条件3>	适用于状态可以 保存和恢复的资源	剥夺次数过多；多次 对资源重新启动
		资源按序申请<条 件4>	可以在编译时（而 不必在运行时）就 进行检查	不便灵活申请新资源
避免(Avoidance)	是“预防”和“检 测”的折衷（在运行 时判断是否可能死锁）	寻找可能的安全 的运行顺序	不必进行剥夺	使用条件：必须知道 将来的资源需求；进程 可能会长时间阻塞
检测(Detection)	宽松的；只要允 许，就分配资源	定期检查死锁是 否已经发生	不延长进程初始化 时间；允许对死锁进 行现场处理	通过剥夺解除死锁， 造成损失

3.6.2 死锁的预防

预防死锁是指通过某种策略来限制并发进程对资源的请求，使系统在任何时刻都不满足死锁

的必要条件。预先静态分配法和有序资源使用法是两种预防死锁的基本策略。针对死锁的第2个条件，预先静态分配法通过预先分配进程运行所需的全部资源，从而保证进程在运行过程中不等待资源。这种做法降低了对资源的利用率，降低了进程的并发程度；并且只适用于有可能预先知道所需资源的情况下，但在实际的进程运行过程中有可能无法预先知道所需的资源。针对死锁的第4个条件，有序资源使用法把资源分类按顺序排列，从而保证资源的申请不形成环路。这种做法会限制进程对资源的请求顺序，同时资源的排序占用系统开销。

3.6.3 死锁的检测

检测死锁的基本思路是在操作系统中保存资源的请求和分配信息，利用某种算法对这些信息加以检查，以判断是否存在死锁。死锁检测算法主要是检查是否有循环等待。我们可把进程和资源间的申请和分配关系描述成一个有向图，通过检查有向图中是否有循环来判断死锁的存在，这就是死锁检测的资源分配图(resource allocation graph)算法。

有向图G的顶点为资源或进程，我们定义从资源R到进程P的边表示R已分配给P，从进程P到资源R的边表示P正因请求R而处于等待状态。如果有向图中存在循环，则表示死锁的存在。为了在复杂的有向图中判断是否有循环，我们可通过资源分配图的等价变换来简化有向图。具体简化过程如下：① 删除不处于等待状态的进程（即没有从该进程出发的边）；② 依次删除当前的叶顶点。可以证明，简化后还存在边的不可简化的资源分配图存在死锁，其中的有边进程为死锁进程。

资源分配图算法可检测死锁，但还需要进一步的操作来解除死锁。解除死锁时，常常造成进程终止或重新启动。我们需要选择或挂起或终止哪一个进程。通常是选择撤消或挂起代价最小的死锁进程，并抢占它的资源，以解除死锁。所谓代价最小的判断原则可为进程优先级或系统会计过程给出的运行代价。

3.6.4 死锁的避免

解决死锁问题的最合理做法应是在分配资源时判断是否会出现死锁，只在确信不会导致死锁时才分配资源。这就是死锁的避免。避免死锁的主要困难在于，我们在很多时候无法判断进程的资源申请是否会导致死锁或判断代价过高。下面我们介绍一种死锁避免算法，即银行家算法。

所谓银行家问题是指银行家在向顾客贷款时如何保证资金的安全。我们假设一个银行家把他的固定资金贷给若干顾客时，只要不出现一个顾客借走所有资金后还不够，银行家的资金就是安全的。银行家需要一个算法保证借出去的资金在有限时间内可收回。

假定顾客分成若干次进行贷款，并在第一次借款时能说明他的最大借款额。在这种假定条件下，银行家算法就可保证资金的安全。具体操作过程如下：顾客的借款操作是依次顺序进行的，直到全部操作完成。银行家对当前顾客的借款操作进行判断：银行能否支持顾客借款，直到全部归还。如果能，则本次贷款是安全的；否则就是不安全的。当判断结果为安全时执行贷款；否则暂不贷款。

由于银行家算法允许资源的部分分配和不需要抢占，因此在操作系统中采用该算法可提高资源利用率。但采用银行家算法的前提太严格，要求事先说明最大资源要求量，这在现实中是很难实现的。

3.6.5 解决死锁问题的综合方法

基于以上分析，我们在实际操作系统中不可能单纯采用某一种方法来解决死锁问题，现实的做法是多种方法的综合使用。例如，首先，我们可对资源进行分类，将各种资源归入若干个不同的资源类中，如外存交换区空间资源、外部设备资源、内存资源等。其次，对资源类进行排序，在不同资源类之间规定次序，对不同资源类中的资源采用线性按序申请的方法。进一步，我们还可对同一资源类中的资源进行针对性处理，采用不同的适当方法。例如，使用避免方法处理外设资源分配，而对存储资源则采用预防方法。

3.7 处理器调度概述

在下面几节中，我们将讨论处理器资源的管理问题，即如何在进程或线程间分配和回收处理器执行时间。处理器是计算机系统中的重要资源，处理器调度算法不仅对处理器的利用效率和用户进程的执行有影响，同时还与内存等其他资源的使用密切相关，对整个计算机系统的综合性能指标有重要影响。

3.7.1 处理器调度的类型

从处理器调度的对象、时间、功能等不同角度，我们可把处理器调度分成不同类型。处理器调度不仅涉及选择哪一个就绪进程进入运行状态，还涉及何时启动一个进程的执行。按照调度所涉及的层次的不同，我们可把处理器调度分成宏观调度、中级调度和微观调度三个层次。

宏观调度也称为作业调度或高级调度。从用户工作流程的角度，一次作业提交若干个流程，其中每个程序按照流程进行调度执行。宏观调度的时间尺度通常是分钟、小时或天。中级调度涉及进程在内外存间的交换。从存储器资源管理的角度来看，把进程的部分或全部换出到外存上，可为当前运行进程的执行提供所需的内存空间，将当前进程所需部分换入到内存。指令和数据必须在内存里才能被处理器直接访问。微观调度也称为低级调度。从处理器资源分配的角度来看，处理器需要经常选择就绪进程或线程进入运行状态。微观调度的时间尺度通常是毫秒级的。由于微观调度算法的频繁使用，因此要求在实现时做到高效。在图3-26中，我们给出了三种层次的处理器调度算法所涉及的进程或线程状态转换。

3.7.2 调度的性能准则

我们可从不同的角度来判断处理器调度算法的性能，如用户的角度、处理器的角度和算法实现的角度。实际的处理器调度算法选择是一个综合的判断结果。

处理器调度是为了执行用户程序，必须考虑用户对调度的要求。用户关心的处理器调度性能指标主要包括周转时间、响应时间、公平性和优先级等。周转时间是指作业从提交到完成（得到结果）所经历的时间。周转时间的组成包括进程在收容队列中的等待时间、占用处理器的执行时间、在就绪队列和阻塞队列中的等待进行等。为了去除进程本身因素的影响，在讨论处理器调度时也使用平均周转时间 T 和平均带权周转时间作为衡量指标。带权周转时间是指周转时间除以进

的运行状况，并在进程出让处理器或调度程序剥夺执行状态进程占用的处理器时，选择适当的进程分派处理器，完成上下文切换。我们把操作系统内核中与进程调度相关代码称为进程调度器(dispatcher)。

进程调度器的主要功能是进程状态维护和进程的上下文切换。当一个进程A进入通过时钟中断或系统调用进入操作系统内核的进程调度器时，首先要保存当前进程A的上下文，执行调度器代码。在调度器代码的控制下确定是否要进行切换，以及切换到哪个进程。如果要切换到另一进程B，则恢复进程B的上下文，然后开始从上次切换前位置继续执行进程B代码。

3.8 调度算法

在这一节中，我们讨论几种主要的处理器调度算法。由于算法的设计出发点不同，因此它们各自的适用场合也不同。有的算法适用于宏观调度，有的算法适用于微观调度，有的算法则是适用于多种场合。

3.8.1 先来先服务算法

先来先服务(First Come First Service, FCFS)算法是最简单的调度算法，它的基本思想是按进程的到达先后顺序进行调度。

FCFS算法按照作业提交或进程变为就绪状态的先后次序来分派处理器；当前作业或进程占用处理器，直到执行完毕或阻塞才让出处理器；当作业或进程唤醒后（如I/O完成），并不立即恢复执行，通常等到当前作业或进程让出处理器才恢复执行。

FCFS算法的最主要特点是简单。由于它的处理器调度采用非抢占方式，因此操作系统不会强行暂停当前进程的执行，FCFS算法具有下列特点：① 比较有利于长作业，而不利于短作业；② 有利于处理器繁忙的作业，而不利于I/O繁忙的作业。

3.8.2 最短作业优先算法

最短作业优先(Shortest Job First, SJF)又称为最短进程优先(Shortest Process Next, SPN)，它的设计目标是改进FCFS算法，减少进程的平均周转时间。SJF算法要求作业在开始执行时预计执行时间，对预计执行时间短的作业（进程）优先分派处理器。后来的短作业不抢先正在执行的作业。

由于SJF算法在分派处理时考虑到进程执行时间对周转时间的影响，因而具有如下优点：① 与FCFS相比，改善了平均周转时间和平均带权周转时间，缩短了作业的等待时间；② 有利于提高系统的吞吐量。但SJF算法也存在一些缺点：① 对长作业非常不利，可能长时间得不到执行；② 未能依据作业的紧迫程度来划分执行的优先级；③ 难以准确估计作业（进程）的执行时间，从而影响调度性能。

通过选用其他条件来分派处理器，SJF算法可形成下列变种：① 最短剩余时间优先(Shortest Remaining Time, SRT)，在SJF算法的基础上，该算法允许比当前进程剩余时间更短的进程来抢占；② 最高响应比优先(Highest Response Ratio Next, HRRN)，响应比的定义为“(等待时间 + 要求执行时间) / 要求执行时间”，它是FCFS和SJF的一种折衷。

3.8.3 时间片时钟算法

前两种算法主要用于宏观调度，说明怎样选择一个进程或作业开始运行，开始运行后的做法都相同，即运行到结束或阻塞，阻塞结束时等待当前进程放弃处理器。时间片时钟算法主要用于微观调度，说明怎样并发运行，即切换的方式；它的设计目标是提高资源利用率。其基本思路是通过时间片轮转，提高进程并发性和响应时间特性，从而提高资源利用率。

在时间片时钟(Round Robin)算法中，系统中所有的就绪进程按照FCFS原则，排成一个队列。每次调度时将处理器分派给队首进程，让其执行一个时间片。时间片的长度从几个ms到几百ms。在一个时间片结束时，发生时钟中断。在时钟中断中，进程调度器暂停当前进程的执行，将其送到就绪队列的末尾，并通过上下文切换执行当前的队首进程。进程可以未使用完一个时间片，就让出处理器（如阻塞）。

时间片时钟算法中的时间片长度是影响算法特征的重要因素。我们先考虑两种极端的情况。如果时间片很长，长到大多数进程可在一个时间片内执行完，该算法将退化为FCFS算法，进程的响应时间长，不能达到提高响应特性的目标。如果时间片过短，用户的一次交互过程也需要多个时间片才能处理完，上下文切换次数增加，响应时间长。因此，时间片长度的选择要与完成一个基本的交互过程所需的时间相当，保证一个基本的交互过程可在一个时间片内完成。我们认为影响时间片长度的主要因素是系统的处理能力和系统的负载状态。依据系统的处理能力确定时间片长度，使用户输入通常在一个时间片内能处理完，否则使响应时间、平均周转时间和平均带权周转时间延长。为了保证不同负载状态下用户交互的响应时间，需要对时间片长度进行适当调整。

3.8.4 多级队列算法

多级队列算法(Multiple-level Queue)的基本思想是引入多个就绪队列，通过各队列的区别对待，达到一个综合的调度目标。在多级队列算法中，根据作业或进程的性质或类型的不同，将就绪队列再分为若干个子队列。每个作业固定归入一个队列，例如，系统进程、用户交互进程、批处理进程等不同队列。不同队列可有不同的优先级、时间片长度、调度策略等。

3.8.5 优先级算法

优先级算法(Priority Scheduling)是多级队列算法的改进，协调各进程队列中进程的响应时间要求。优先级算法适用于作业调度和进程调度，可分成抢先式和非抢先式。

优先级算法中各进程的优先级确定方式分为静态和动态两种。静态优先级方式是指在创建进程时确定进程优先级，并保持不变到进程结束。影响进程的静态优先级的主要因素包括进程类型、进程的资源需求和用户要求。通常，系统进程优先级高于其他用户进程，对处理器和内存等资源要求较少的进程优先级较高。进程也可按用户的紧急程度和付费情况等来确定。动态优先级方式是指在创建进程时赋予给进程的优先级，在进程运行过程中可以自动改变，以便获得更好的调度性能。影响进程动态优先级变化的因素包括进程等待时间和占用处理器时间等。当一个进程在就

就绪队列中等待时间越长，它的优先级会越高。这种做法的目的是使优先级较低的进程在等待足够的时间后，其优先级提高，进而被调度执行。当一个进程占用处理器的执行的时间越长，它的优先级就会越低。这种做法的目的是使持续执行的进程会在优先级降低后出让处理器。

3.8.6 多级反馈队列算法

多级反馈队列(Round Robin with Multiple Feedback)算法是时间片时钟算法和优先级算法的综合和发展。通过动态调整进程优先级和时间片大小，多级反馈队列算法可兼顾多方面的系统目标。例如，为提高系统吞吐量和缩短平均周转时间而照顾短进程；为获得较好的I/O设备利用率和缩短响应时间而照顾I/O型进程；同时，也不必事先估计进程的执行时间。

在多级反馈队列算法中，通常设置有多个就绪队列，分别赋予不同的优先级，如队列1的优先级最高，然后逐级降低。每个队列的执行时间片长度也不同，如规定优先级越低则时间片越长。新进程进入内存后，先投入最高优先级的队列1的末尾，按FCFS算法调度；如果队列1的一个时间片未能执行完，则降低投入到队列2的末尾，同样按FCFS算法调度；如此下去，一直降低到最后的队列，则按时间片时钟算法调度直到完成。仅当较高优先级的队列为空时，才调度较低优先级的队列中的进程执行。如果进程执行时有新进程进入较高优先级的队列，则抢先执行新进程，并把被抢先的进程投入原队列的末尾。

在实际系统中使用的多级反馈队列算法还可以使用更复杂的动态优先级调整策略。例如，为了保证I/O操作的及时完成，通常会在进程发出I/O请求后进入最高优先级队列，并执行一个时间片，以及时响应I/O交互。对于计算型进程，可在每次执行完一个完整的时间片后，进入更低级队列，并最终采用最大时间片来执行，这样可减少计算型进程的调度次数。对于I/O次数不多而处理器占用时间较多的进程，可使用高优先级进行I/O处理，在I/O完成后，放回原来队列，以免每次都从最高优先级队列逐次下降。一种更通用的策略是：进行I/O时提高优先级；时间片用完时降低优先级。这样可适应一个进程在不同时间段的运行特点。

3.9 Windows 2000/XP的线程调度

作为一个实际的操作系统，Windows 2000/XP的处理器调度的调度对象是线程，也称为线程调度。作为一个实际的操作系统，Windows 2000/XP的线程调度并不是单纯使用某一种调度算法，而是多种算法的结合体，根据实际系统的需要进行针对性的优化和改进。下面从Windows 2000/XP中与处理器调度相关的各个侧面进行深入讨论。首先简要描述Windows 2000/XP线程调度的主要特征；然后从Win32应用程序编程接口和Windows 2000/XP内核的角度讨论优先级；最后说明Windows 2000/XP线程调度的数据结构和调度算法。

3.9.1 Windows 2000/XP的线程调度特征

Windows 2000/XP实现了一个基于优先级的抢先式多处理器调度系统。调度系统总是运行优先级最高的就绪线程。通常线程可在任何可用处理器上运行，但可限制某线程只能在某处理器上运行。亲和处理器集合允许用户线程通过Win32调度函数选择它偏好的处理器。

当一个线程被调度进入运行状态时，它可运行一个被称为时间配额(quantum)的时间片。时间配额是Windows 2000/XP允许一个线程连续运行的最大时间长度，随后Windows 2000/XP会中断该线程的运行，判断是否需要降低该线程的优先级，并查找是否有其他高优先级或相同优先级的线程等待运行。Windows 2000专业版和Windows 2000服务器版的时间配额是不同的，同一系统中各线程的时间配额是可修改的。由于Windows 2000/XP的抢先式调度特征，因此一个线程的一次调度执行可能并没有用完它的时间配额。如果一个高优先级的线程进入就绪状态，当前运行的线程可能在用完它的时间配额前就被抢先。事实上，一个线程甚至可能在被调度进入运行状态之后开始运行之前就被抢先。

Windows 2000/XP在内核中实现它的线程调度代码，这些代码分布在内核中与调度相关事件出现的位置，并不存在一个单独的线程调度模块。内核中完成线程调度功能的这些函数统称为内核调度器(kernel's dispatcher)。线程调度出现在DPC/线程调度中断优先级。线程调度的触发事件有以下四种：

- 1) 一个线程进入就绪状态，如一个刚创建的新线程或一个刚刚结束等待状态的线程。
- 2) 一个线程由于时间配额用完而从运行状态转入退出状态或等待状态。
- 3) 一个线程由于调用系统服务而改变优先级或被Windows 2000/XP系统本身改变其优先级。
- 4) 一个正在运行的线程改变了它的亲和处理器集合。

在这些触发事件出现时，Windows 2000/XP必须选择下一个要运行的线程。当Windows 2000/XP选择一个新线程进入运行状态时，将执行一个线程上下文切换以使新线程进入运行状态。线程上下文是指保存正在运行线程的相关运行环境，加载另一个线程的相关运行环境，并开始新线程执行的过程。

我们已说过，Windows 2000/XP的处理器调度对象是线程，这时的进程仅作为提供资源对象和线程的运行环境，而不是处理器调度的对象。处理器调度是严格针对线程进行的，并不考虑被调度线程属于哪个进程。例如，进程A有10个可运行的线程，进程B有2个可运行的线程，这12个线程的优先级都相同，则每个线程将得到1/12的处理器时间。Windows 2000/XP并不会把处理器时间分成相同的两半，一半给进程A，另一半给进程B。

为了理解线程调度算法，我们首先要说明Windows 2000/XP所使用的优先级。

3.9.2 Win32中与线程调度相关的应用程序编程接口

表3-6中给出了Win32 API中与线程调度相关的函数列表。更详细的信息可参见Win32 API的参考文档。

表 3-6

与线程调度相关的API函数名	函数功能
Suspend/ResumeThread	挂起一个正在运行的线程或激活一个暂停运行的线程
Get/SetPriorityClass	读取或设置一个进程的基本优先级类型
Get/SetThreadPriority	读取或设置一个线程相对优先级(相对进程优先级类型)

(续)

与线程调度相关的API函数名	函数功能
Get/SetProcessAffinityMask	读取或设置一个进程的亲和处理器集合
SetThreadAffinityMask	设置线程的亲和处理器集合(必须是进程亲和处理器集合的子集), 只允许该线程在指定的处理器集合运行
Get/SetThreadPriorityBoost	读取或设置Windows 2000/XP暂时提升线程优先级状态; 只能在可调范围内提升
SetThreadIdealProcessor	设置一个特定线程的首选处理器; 不限制该线程只能在该处理器上运行
Get/SetProcessPriorityBoost	读取或设置当前进程的缺省优先级提升控制。该功能用于在创建线程时控制线程优先级的暂时提升状态
SwitchToThread	当前线程放弃一个或多个时间配额的运行
Sleep	使当前线程等待指定的一段时间(时间单位为毫秒)。0表示放弃该线程的剩余时间配额
SleepEx	使当前线程进入等待状态, 直到I/O处理完成、有一个与该线程相关的APC或经过一段指定的时间

3.9.3 线程优先级

如图3-27所示, Windows 2000/XP内部使用32个线程优先级, 范围从0到31, 它们被分成以下三个部分。

- 1) 16个实时线程优先级 (16 ~ 31)。
- 2) 15个可变线程优先级 (1 ~ 15)。
- 3) 一个系统线程优先级 (0), 仅用于对系统中空闲物理页面进行清零的零页线程。

线程优先级的指定可从两个不同的角度进行: 用户可通过Win32应用程序编程接口来指定线程的优先级, Windows 2000/XP内核也可控制线程的优先级。Win32应用程序编程接口可在进程创建时指定其优先级类型为实时、高级、中上、中级、中下和空闲, 并进一步在进程内各线程创建时指定线程的相对优先级为相对实时、相对高级、相对中上、相对中级、相对中下、相对低级和相对空闲。

通过Win32应用程序编程接口指定的线程优先级是由进程优先级类型和线程相对优先级共同控制。图3-28给出Win32线程优先级到Windows 2000/XP内部优先级的映射关系。

图3-28给出的是线程的基本优先级。通过任务管理器 (Task Manager) 或Win32应用程序编程接口的SetPriorityClass函数可指定进程的基本优先级, 线程从继承的进程基本优先级开始运行。

进程基本优先级和线程开始时的优先级通常是缺省地设置为各进程优先级类型的中间值 (24、13、10、8、6或4)。Windows 2000/XP的一些系统进程 (如会话管理器、服务控制器和本地安全

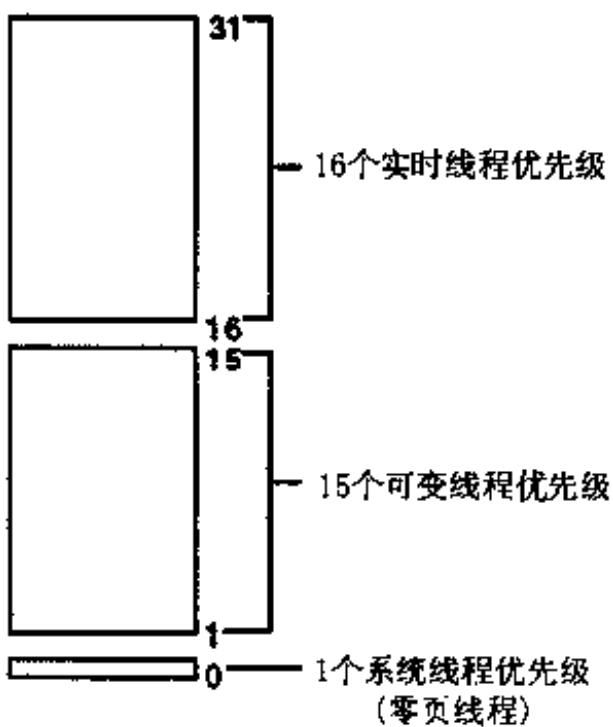


图3-27 线程优先级

认证服务器等)的基本优先级比缺省的中级(8)要高一些。这样可保证这些进程中的线程在开始时就具有高于缺省值8的优先级。系统进程可使用Windows 2000/XP的内部函数来设置比Win32基本优先级更高的进程基本优先级。

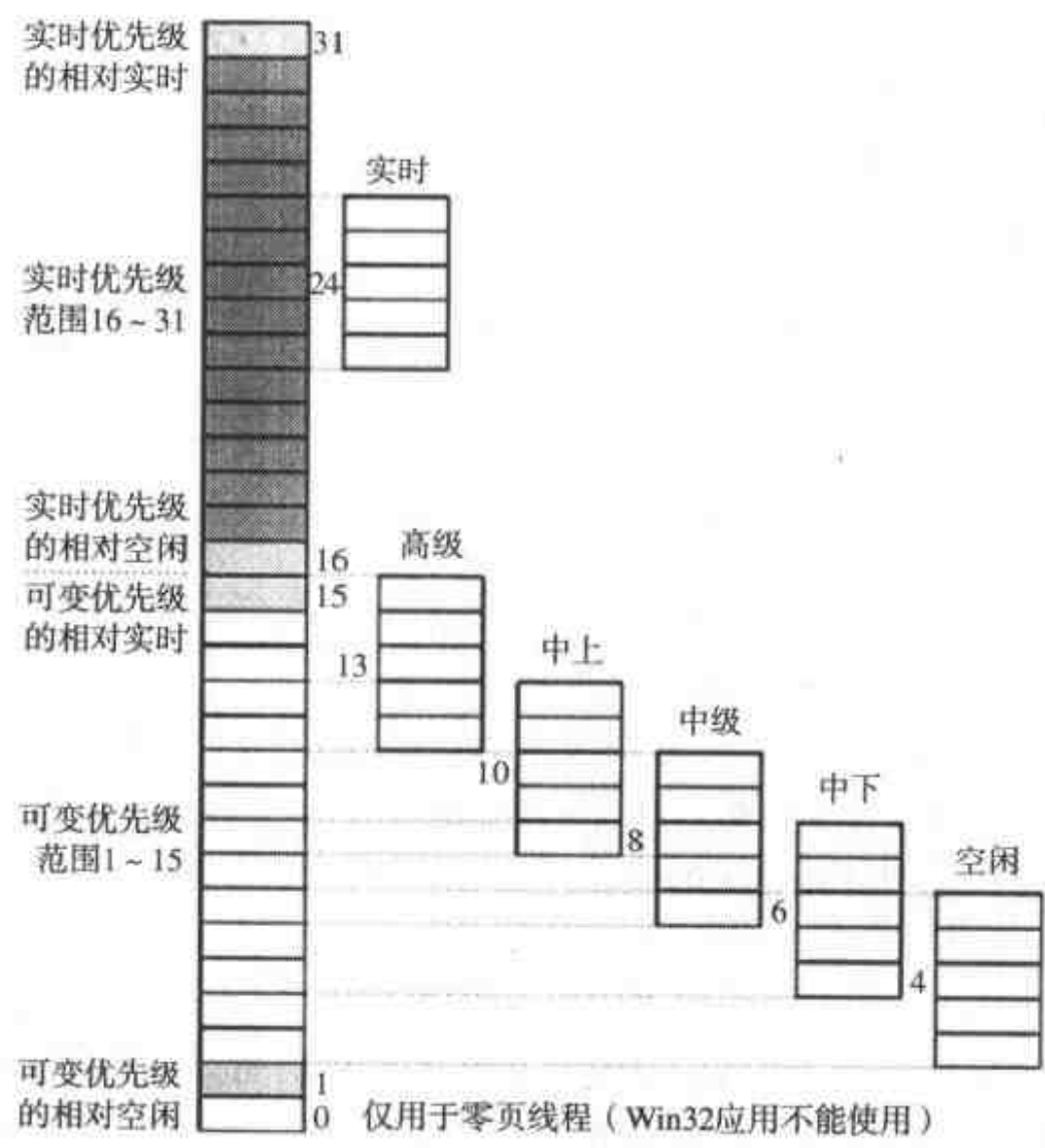


图3-28 Win32和Windows 2000/XP的线程优先级

一个进程仅有单个优先级取值(基本优先级)，而一个线程有当前优先级和基本优先级这两个优先级取值。线程的当前优先级可在一定范围(1至15)内动态变化，通常会比基本优先级高。Windows 2000/XP从不调整在实时范围(16至31)内的线程优先级，因而这些线程的基本优先级和当前优先级总是一样的。

1. 实时优先级

在应用程序中，用户可在一定范围内升高或降低线程优先级。要把线程的优先级提升到实时优先级，用户必须有升高线程优先级的权限。如果用户试图提升进程优先级到实时类型，但没有相应权限，则相应操作并不会失败，而是提升到高级类型。

我们知道，Windows 2000/XP有许多重要内核系统线程是运行在实时优先级的。如果用户进程在实时优先级运行时间过多，它将可能阻塞关键系统功能(如存储管理器、缓存管理器、本地和网络文件系统、甚至设备驱动程序等)的执行，阻塞系统线程的运行；但由于硬件中断的优先级比任何线程都要高，因此它不会阻塞硬件中断处理。

在被其他线程抢先时，具有实时优先级的线程的行为与具有可变优先级的线程的行为是不同

的。具有实时优先级的线程在被抢先时，它的时间配额将会被重置成进入运行状态时的初值。

注：虽然Windows 2000/XP有一组优先级称为实时优先级，但是它们并不是通常意义上的实时。Windows 2000/XP并不提供实时操作系统服务，如有保证的中断处理延时或确保线程得到一定执行时间的机制。

2. 中断优先级与线程优先级的关系

如图3-29所示，所有线程都运行在中断优先级0和1。用户态线程运行在中断优先级0，内核态的异步过程调用运行在中断优先级1，它们会中断线程的运行。只有内核态线程可提升自己的中断优先级；虽然高优先级的实时线程可阻塞重要的系统线程执行，但不管用户态线程的优先级是多少，它都不会阻塞硬件中断。

线程调度代码是运行在DPC/线程调度中断优先级的。因此，当内核正在选择下一个要运行的线程时，系统中不会有线程正在运行，也不可能修改线程优先级等与调度相关的信息。在多处理器系统中，访问线程调度数据结构是通过请求线程调度器自旋锁(KiDispatcherLock)来实现同步的。

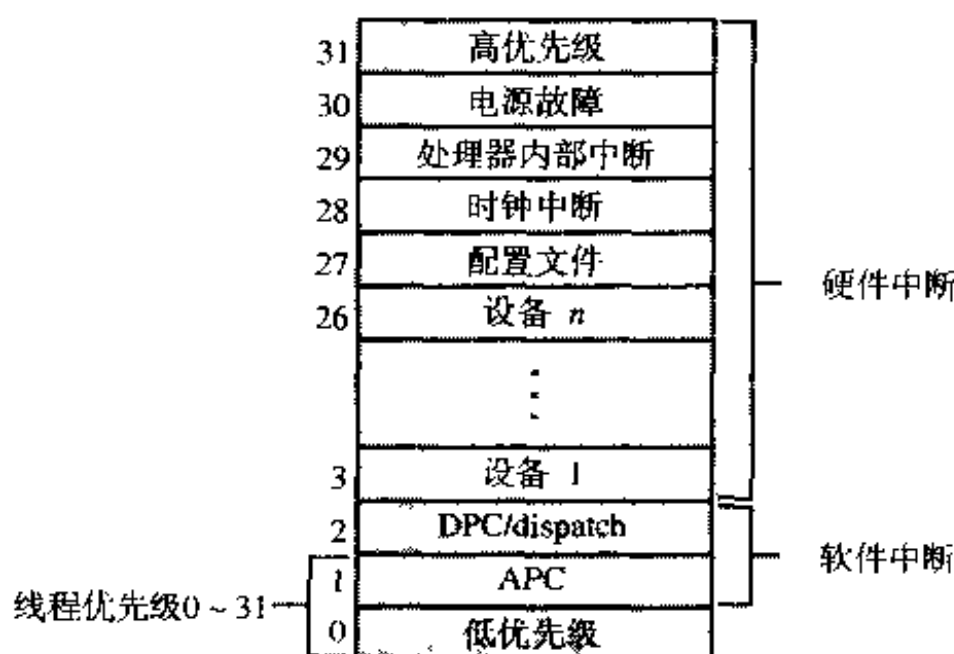


图3-29 中断优先级与线程优先级

3.9.4 线程时间配额

正如前面所述，时间配额是一个线程从进入运行状态到Windows 2000/XP检查是否有其他优先级相同的线程需要开始运行之间的时间总和。一个线程用完了自己的时间配额时，如果没有其他相同优先级的线程，Windows 2000/XP将重新给该线程分配一个新的时间配额，并继续运行。

每个线程都有一个代表本次运行最大时间长度的时间配额。时间配额不是一个时间长度值，而是一个称为配额单位(quantum unit)的整数。

1. 时间配额的计算

缺省时，在Windows 2000专业版中线程开始时的时间配额为6；而在Windows 2000/XP服务器版中线程开始时的时间配额为36。后面我们将介绍如何修改缺省时间配额值。在Windows 2000/XP服务器版中取较长缺省时间配额的原因是要保证客户请求所唤醒的服务器应用有足够的时间在它的时间配额用完前完成客户的请求并回到等待状态。

每次时钟中断，时钟中断服务例程从线程的时间配额中减少一个固定值(3)。如果没有剩余的时间配额，系统将触发时间配额用完处理，选择另外一个线程进入运行状态。在Windows 2000专业版中，由于每个时钟中断时减少的时间配额为3，因此一个线程的缺省运行时间为2个时钟中断间隔；在Windows 2000/XP服务器版中，一个线程的缺省运行时间为12个时钟中断间隔。

如果时钟中断出现时系统正处在DPC/线程调度中断优先级以上(如系统正在执行一个DPC或一个中断服务例程)，当前线程的时间配额仍然要减少。甚至在整个时钟中断间隔期间，当前线

程一条指令也没有执行，它的时间配额在时钟中断中也会被减少。

不同硬件平台的时钟中断间隔是不同的，时钟中断的频率是由硬件抽象层确定的，而不是由内核确定的。例如，大多数x86单处理器系统的时钟中断间隔为10毫秒，大多数x86多处理器系统的时钟中断间隔为15毫秒。

利用Win32的函数GetSystemTimeAdjustment可得到系统的时钟中断间隔。用www.sysinternals.com上的Clockres程序也可得到系统的时钟中断间隔。

在一个空闲的系统上，利用性能监视工具也可大致估计系统的时钟中断间隔。空闲系统是指没有进程在执行I/O操作；通过检查每个进程的I/O计数，可验证系统是否空闲。观察性能监视工具中的处理器对象中的每秒中断次数计数(Interrupts/sec)，该计数平均值的倒数就是系统的时钟中断间隔。

例如，在大多数的x86单处理器系统中，每秒中断次数计数的平均值为100，因此可计算出时钟中断间隔为1/100=0.01秒，即10毫秒。在一个x86多处理器系统中，每秒中断次数计数的平均值为67，则时钟中断间隔为1/67=0.015秒，即15毫秒。

用3个时间配额单位，而一个时间配额单位表示一个时钟中断间隔的目的是，在等待完成时允许减少部分时间配额。当优先级小于14的线程执行一个等待函数(如WaitForSingleObject或WaitForMultipleObjects)时，它的时间配额被减少1个时间配额单位。当优先级大于等于14的线程执行完等待函数后，它的时间配额被重置。

这种部分减少时间配额的做法可解决线程在时钟中断触发前进入等待状态所产生的问题。如果不进行这种部分减少时间配额的操作，一个线程可能永远不会减少它的时间配额。例如，一个线程运行一段时间后进入等待状态，再运行一段时间后又进入等待状态，但在时钟中断出现时它都不是当前线程，则它的时间配额永远也不会因为运行而减少。

2. 时间配额的控制

在系统注册库中的一个注册项“HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation”，允许用户指定线程时间配额的相对长度(长或短)和前台进程的时间配额是否加长。如图3-30所示，该注册项为6位，分成3个字段，每个字段占2位。

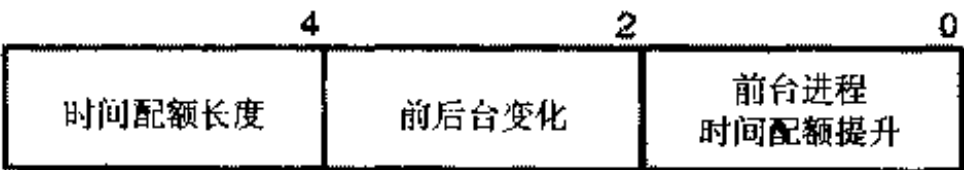


图3-30 注册项Win32PrioritySeparation的定义

时间配额长度字段(Short vs. Long): 1表示长时间配额；2表示短时间配额；0或3表示缺省设置(Windows 2000专业版的缺省设置为短时间配额，Windows 2000服务器版的缺省设置为长时间配额)。

前后台变化字段 (Variable vs. Fixed): 1表示改变前台进程时间配额；2表示前后台进程的时间配额相同；0或3表示缺省设置(Windows 2000专业版的缺省设置为改变前台进程时间配额，Windows 2000服务器版的缺省设置为前后台进程的时间配额相同)。

前台进程时间配额提升字段(Foreground Quantum Boost): 该字段的取值只能是0、1或2(取3是非法的, 被视为2)。该字段是一个时间配额表索引, 用于设置前后台进程的时间配额, 后台进程的时间配额为第一项, 前台进程的时间配额为第二项。该字段的值保存在内核变量PsPrioritySeparation。

前台进程是指拥有屏幕当前窗口的线程所在的进程。如果当前窗口切换到一个优先级高于空闲优先级类的进程中的某个线程, Win32子系统将用注册项Win32PrioritySeparation的前台进程时间配额字段作为索引, 依据一个有3个元素的数组PspForegroundQuantum中的取值, 来设置该进程中所有线程的时间配额。该数组的内容由注册项Win32PrioritySeparation的另外2个字段确定。表3-7给出了数组PspForegroundQuantum的可能时间配额设置。

表 3-7

	短时间配额			长时间配额		
改变前台进程时间配额	6	12	18	12	24	36
前后台进程的时间配额相同	18	18	18	36	36	36

下面我们举例说明, 为什么在Windows 2000/XP中要增加前台线程的时间配额, 而不是提高前台线程的优先级。基于提高前台线程优先级的做法存在一个潜在的问题。假设用户首先启动了一个运行时间很长的电子表格计算程序, 然后切换到一个计算密集型的应用(如一个需要复杂图形显示的游戏)。如果前台的游戏进程提高它的优先级, 后台的电子表格将会几乎得不到处理器时间。但增加游戏进程的时间配额, 则不会停止电子表格计算的执行, 而只是给游戏进程的处理器时间多一些。如果用户希望运行一个交互式应用程序时的优先级比其他交互进程的优先级高, 可利用任务管理器将进程的优先级类型修改为中上或高级, 也可利用命令行在启动应用时使用命令“start /abovenormal”或“start /high”来设置进程优先级类型。在任务管理器中修改进程优先级类型的方法为, 在“进程”栏中用鼠标右键激活下拉菜单中的“设置优先级”。

通过直接修改注册项Win32PrioritySeparation来设置时间配额时, 用户可对3个字段中的内容进行任意组合。通过控制面板中的性能设置(Performance Options)窗口来设置时间配额时, 你只有2种选择。用户可从“控制面板”中的“系统”或“我的电脑”中的“属性”里找到“性能”设置窗口。

为了优化应用的性能, 时间配额的设置可为短时间配额和改变前台进程的时间配额, 这是Windows 2000专业版的缺省设置。为了优化后台服务的性能, 设置可为长时间配额和前后台进程的时间配额相同, 这是Windows 2000服务器版的缺省设置。如果在Windows 2000高级服务器或Windows 2000数据中心服务器上安装远程终端服务(Terminal Services), 并且配置该服务器为应用服务器时, 时间配额设置为优化应用的性能。

3.9.5 调度数据结构

如图3-31所示, 为了进行线程调度, 内核维护了一组称为“调度器数据结构”的数据结构。调度器数据结构负责记录各线程的状态, 如哪些线程处于等待状态、处理器正在执行哪个线程等。

调度器数据结构中的最主要内容是调度器的就绪队列(KiDispatcherReadyListHead)，该队列由一组子队列组成，每个调度优先级有一个队列，其中包括该优先级的等待调度执行的就绪线程。

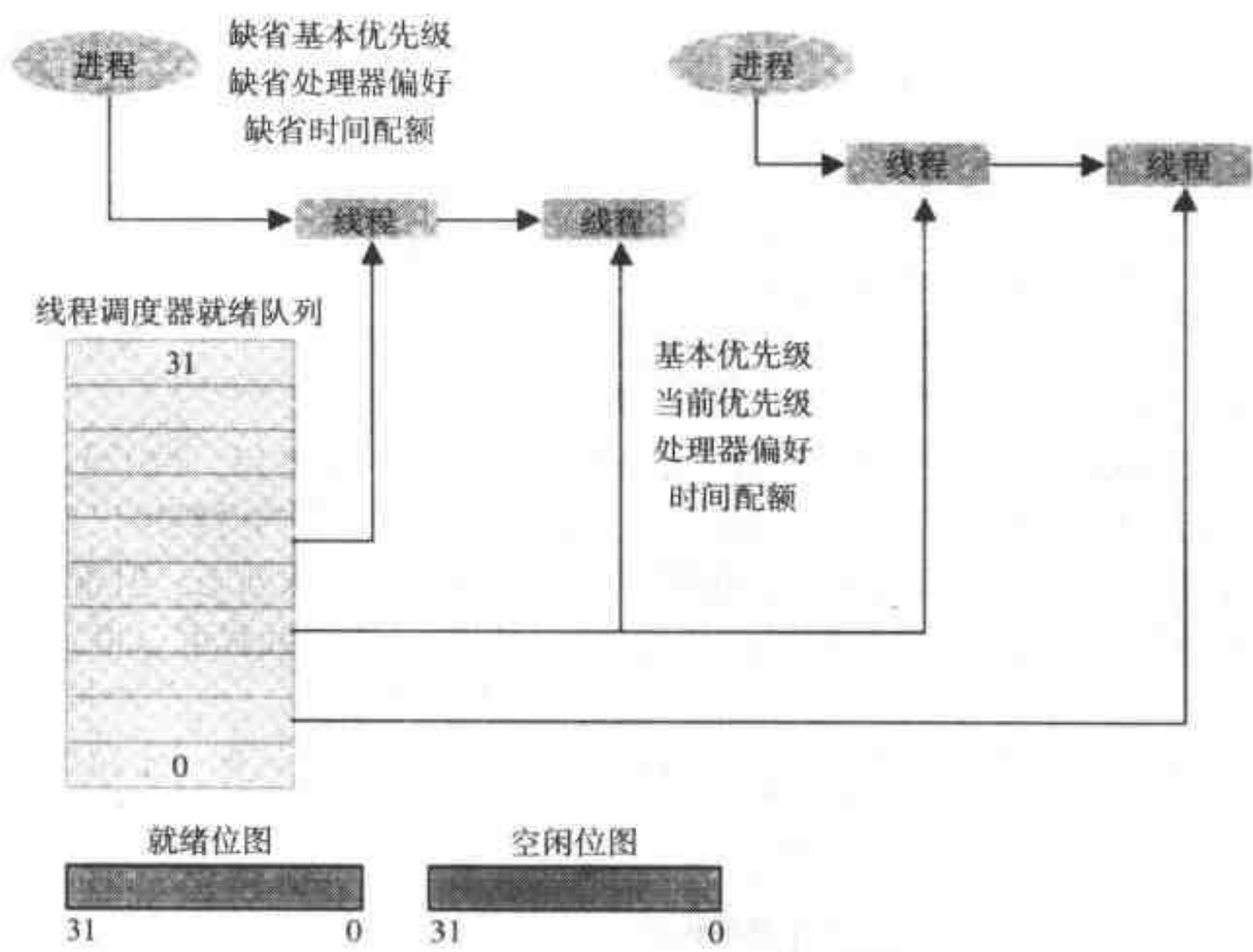


图3-31 线程调度器数据结构

为了提高调度速度，Windows 2000/XP维护了一个称为就绪位图(KiReadySummary)的32位量。就绪位图中的每一位指示一个调度优先级的就绪队列中是否有线程等待运行。B0与调度优先级0相对应，B1与调度优先级1相对应，等待。Windows 2000/XP还维护一个称为空闲位图(KiIdleSummary)的32位量。空闲位图中的每一位指示一个处理器是否处于空闲状态。

如前所述，为了防止调度器代码与线程在访问调度器数据结构时发生冲突，线程调度仅出现在DPC/线程调度中断优先级。但在多处理器系统中，修改调度器数据结构需要额外的步骤来得到内核调度器自旋锁(KiDispatcherLock)，以协调各处理器对调度器数据结构的访问。表3-8给出了与线程调度相关的内核变量。

表 3-8

变 量 名	变 量 类 型	功 能 说 明
KiDispatcherLock	自旋锁	调度器自旋锁
KeNumberProcessors	字节	系统中的可用处理器数目
KeActiveProcessors	32位位图	描述系统中各处理器是否正处于运行状态
KiIdleSummary	32位位图	描述系统中各处理器是否处于空闲状态
KiReadySummary	32位位图	描述各调度优先级是否有就绪线程等待调度
KiDispatcherReadyListHead	有32个元素的数组	32个元素分别指向32个就绪队列

3.9.6 调度策略

Windows 2000/XP严格基于线程的优先级来确定哪一个线程将占用处理器并进入运行状态。但在实际系统中是如何实现的？下面章节将说明如何基于线程实现优先级驱动的抢先式多任务。需要说明的是，Windows 2000/XP在单处理器系统和多处理器系统中的线程调度是不同的。这里我们首先介绍单处理器系统中的线程调度。

1. 主动切换

首先一个线程可能因为进入等待状态而主动放弃处理器的使用。许多Win32等待函数调用(如WaitForSingleObject或WaitForMultipleObjects等)都使线程等待某个对象，等待的对象可能有事件、互斥信号量、资源信号量、I/O操作、进程、线程、窗口消息等。

主动切换可比喻成一个线程在快餐柜台买了一份还未完成的汉堡包。为了不阻塞它后面的就餐者购买快餐，它可以先站在一边等待，以便下一个线程在它等待的时候可以购买(运行它的例程)。当该线程等待的汉堡包做好时，它会排到相应优先级的就绪队列尾。但读者在后面可以见到，大多数的等待操作都会导致临时性的优先级提高，以便让等待线程可以得到它买的汉堡包，并开始就餐。

图3-32说明了在一个线程进入等待状态时Windows 2000/XP如何选择一个新线程开始运行。

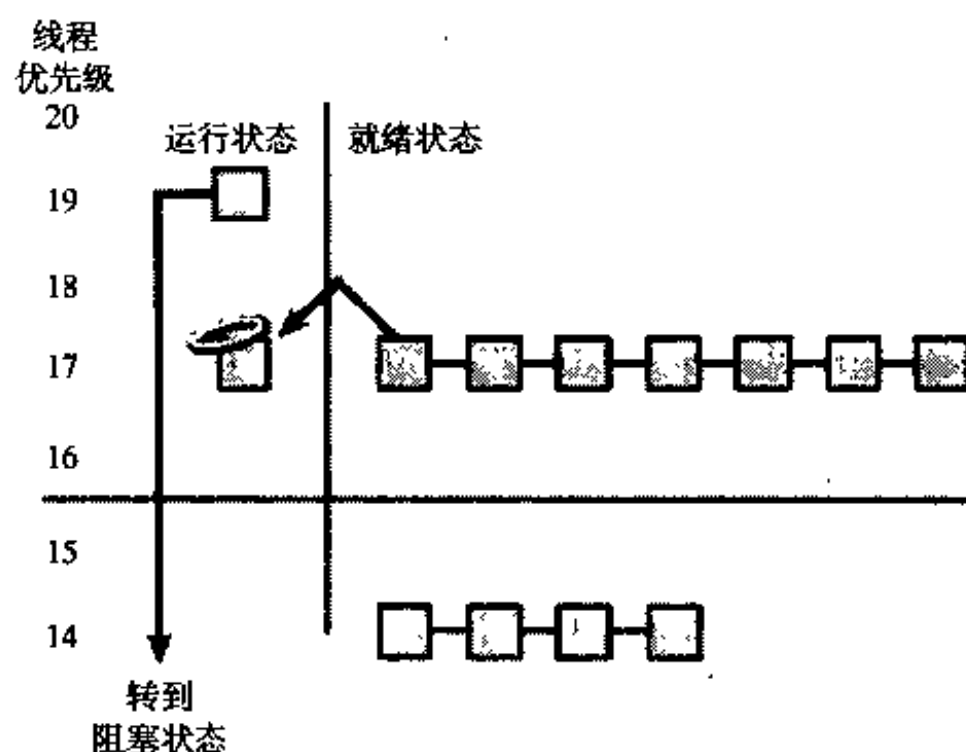


图3-32 主动切换

在图3-32中，正方形表示线程，最上方的线程主动放弃处理器占用，就绪队列中的第一个线程(带光环的正方形)进入运行状态。虽然图3-32中主动放弃处理器的线程被降低了优先级，但这并不是必须的，可以仅仅是被放入等待对象的等待队列中。如何处理线程的剩余时间配额？通常进入等待状态线程的时间配额不会被重置，而是在等待事件出现时，线程的时间配额被减1，相当于1/3个时钟间隔；如果线程的优先级大于等于14，在等待事件出现时，线程的优先级被重置。

2. 抢先

在这种情况下，当一个高优先级线程进入就绪状态时，正在处于运行状态的低优先级线程被

抢先。可能在以下两种情况下出现抢先：

- 1) 高优先级线程的等待完成，即一个线程等待的事件出现。
- 2) 一个线程的优先级被增加或减少。

在这两种情况下，Windows 2000/XP都要确定是否让当前线程继续运行或是否当前线程要被一个高优先级线程抢先。

注：用户态下运行的线程可以抢先内核态下运行的线程。在判断一个线程是否被抢先时，并不考虑线程处于用户态还是内核态，调度器只是依据线程优先级进行判断。

当线程被抢先时，它被放回相应优先级的就绪队列的队首。处于实时优先级的线程在被抢先时，时间配额被重置为一个完整的时间片；而处于动态优先级的线程在被抢先时，时间配额不变，重新得到处理器使用权后将运行到剩余的时间配额用完。图3-33说明了线程抢先的过程。

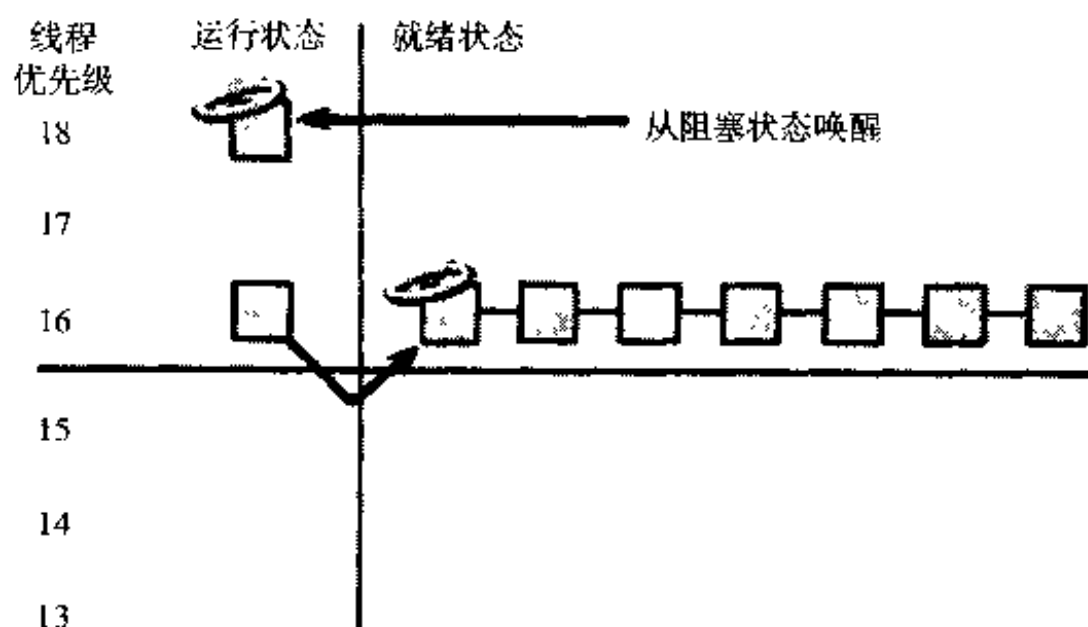


图3-33 线程的抢先调度

在图3-33中，一个优先级为18的线程从等待状态返回并收复处理器，这导致优先级为16的正在运行的线程被弹回到就绪队列的队首。注意，被抢先的线程是排在就绪队列的队首，而不是队尾。当抢先线程完成运行后，被抢先的线程可继续它的剩余时间配额。在这个例子中，线程的优先级都在实时优先级范围，它们的优先级不会被动态提升。

如果我们把主动切换比喻成一个线程在它等待自己的汉堡包时允许排在它后面的线程可以买快餐，抢先则可比喻成由于美国总统来到快餐店要求买快餐，一个正在运行的线程被挤回到就绪队列。被抢先的线程并不是排到就绪队列的队尾，而只是在总统买快餐时站在一旁；一旦总统离开，它会恢复运行，完成快餐采购。

3. 时间配额用完

当一个处于运行状态的线程用完它的时间配额时，Windows 2000/XP首先必须确定是否需要降低该线程的优先级，然后确定是否需要调度另一个线程进入运行状态。

如果刚用完时间配额的线程的优先级被降低了，Windows 2000/XP将寻找一个更适合的线程进入运行状态；所谓更适合的线程是指优先级高于刚用完时间配额的线程的新设置值的就绪线程。如果刚用完时间配额的线程的优先级没有降低，并且有其他优先级相同的就绪线程，

Windows 2000/XP将选择相同优先级的就绪队列中的下一个线程进入运行状态，刚用完时间配额的线程被排到就绪队列的队尾(即分配一个新的时间配额并把线程状态从运行状态改为就绪状态)。图3-34说明了这个调度过程。如果没有优先级相同的就绪线程可运行，刚用完时间配额的线程将得到一个新的时间配额并继续运行。

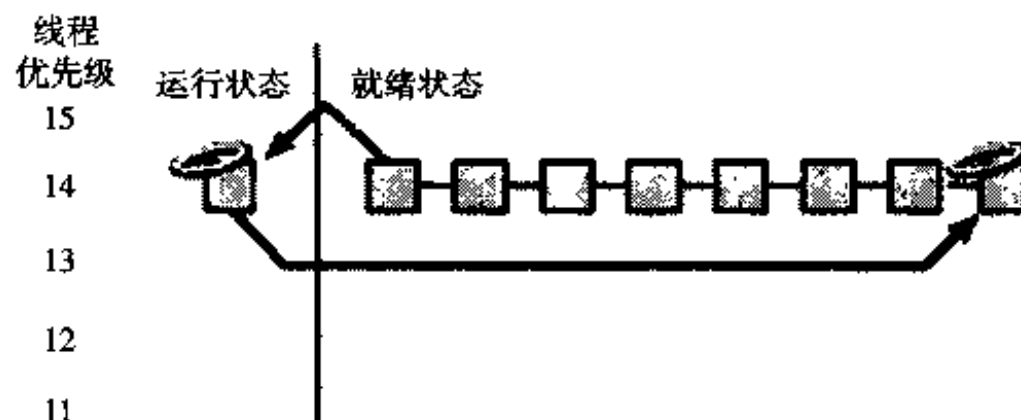


图3-34 时间配额用完时的线程调度

4. 结束

当线程完成运行时，它的状态从运行状态转到终止状态。线程完成运行的原因可能是通过调用ExitThread而从主函数中返回或被其他线程通过调用TerminateThread来终止。如果处于终止状态的线程对象上没有未关闭的句柄，则该线程将被从进程的线程列表中删除，相关数据结构将被释放。

3.9.7 线程优先级提升

在下列5种情况下，Windows 2000/XP会提升线程的当前优先级：

- 1) I/O操作完成。
- 2) 信号量或事件等待结束。
- 3) 前台进程中的线程完成一个等待操作。
- 4) 由于窗口活动而唤醒图形用户接口线程。
- 5) 线程处于就绪状态超过一定时间，但没能进入运行状态(处理器饥饿)。

其中，前两条是针对所有线程进行的优先级提升，而后三条是针对某些特殊的线程在正常的优先级提升基础上进行额外的优先级提升。线程优先级提升的目的是改进系统吞吐量、响应时间等整体特征，解决线程调度策略中潜在的不公正性。与任何调度算法一样，线程优先级提升也不是完美的，它并不会使所有应用都受益。

注：Windows 2000/XP永远不会提升实时优先级范围内(16至31)的线程优先级。因此，在实时优先级范围内的线程调度总是可以预测的。在使用实时优先级时，Windows 2000/XP假定用户完全了解线程的行为。

1. I/O操作完成后的线程优先级提升

在完成I/O操作后，Windows 2000/XP将临时提升等待该操作线程的优先级，以保证等待I/O操作的线程能有更多的机会立即开始处理得到的结果。如前所述，为了避免I/O操作导致对某些

线程的不公平偏好，在I/O操作完成后唤醒等待线程时将把该线程的时间配额减1。虽然在DDK头文件中有关于优先级提升的建议值，但线程优先级的实际提升值是由设备驱动程序决定的。与I/O操作相关的线程优先级提升建议值在文件“Wdm.h”或“Ntddk.h”中以“#define IO”串开始处；表3-9是线程优先级提升建议值的列表。设备驱动程序在完成I/O请求时通过内核函数IoCompleteRequest来指定优先级提升的幅度。

注：在表3-9中，线程优先级的提升幅度与I/O请求的响应时间要求是一致的，响应时间要求越高，优先级提升幅度越大。

表 3-9

设 备	优先级提升幅度
磁盘、光驱、并口、视频	1
网络、邮件槽、命名管道、串口	2
键盘、鼠标	6
音频	8

线程优先级提升是以线程的基本优先级为基点的，不是以线程的当前优先级为基点。如图3-35所示，线程优先级提升后将在提升后的优先级上运行一个时间配额。当用完它的一个时间配额后，线程会降低一个优先级，并运行另一个时间配额。这个降低过程会一直进行下去，直到线程的优先级降低至原来的基本优先级。其他优先级较高的线程仍可抢先因I/O操作而提升了优先级的线程，但被抢先的线程要在提升后的优先级上用完它的时间配额后才降低一个优先级。

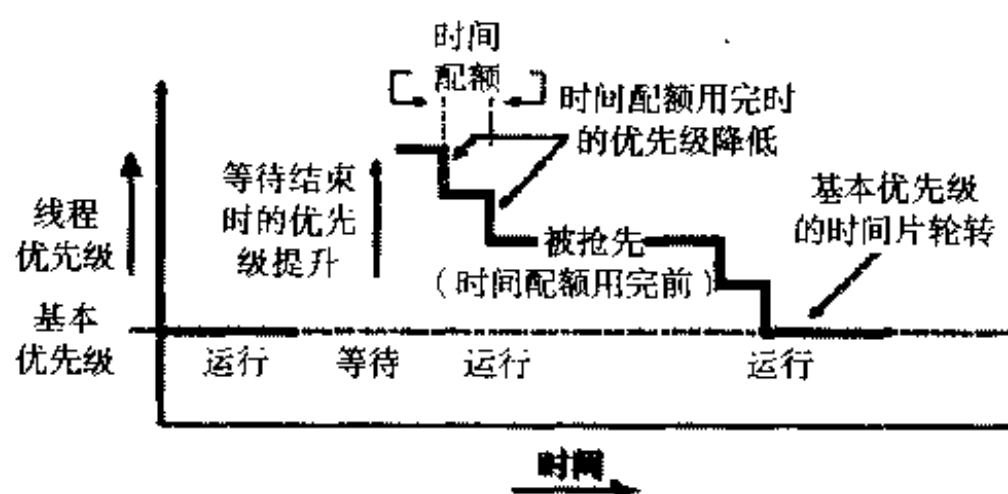


图3-35 线程优先级的提升和降低

如前所述，优先级提升策略仅适用于可变优先级范围(0到15)内的线程。不管线程的优先级提升幅度有多大，提升后的优先级都不会超过15而进入实时优先级。也就是说，一个优先级为14的线程要提升5个优先级，则提升后的线程优先级为15；一个优先级为15的线程提升优先级后的线程优先级还是15。

2. 等待事件和信号量后的线程优先级提升

当一个等待执行事件对象或信号量对象的线程完成等待后，它的线程将提升一个优先级。SetEvent、PulseEvent或ReleaseSemaphore函数调用可导致事件对象或信号量对象等待的结束。

在DDK头文件中定义的常量EVENT_INCREMENT和SEMAPHORE_INCREMENT表示事件和信号量等待结束后的优先级提升幅度。线程在等待事件或信号量后提升优先级的原因与等待I/O操作后的优先级提升的原因类似。阻塞于事件或信号量的线程得到的处理器时间比处理器繁忙型线程要少，这种提升可减少这种不平衡带来的影响。

等待事件或信号量后的线程优先级提升与I/O操作完成时的提升相同：提升是以线程的基本优先级为基点的，而不是以线程的当前优先级为基点的。提升后的优先级永远不会超过15。在等待结束时，线程的时间配额被减1，并在提升后的优先级上执行完剩余的时间配额；随后降低1个优先级，运行一个新的时间配额，直到优先级降低到初始的基本优先级。

3. 前台线程在等待结束后的优先级提升

对于前台进程中的线程，一个内核对象上的等待操作完成时，内核函数KiUnwaitThread会提升线程的当前优先级(不是线程的基本优先级)，提升幅度为变量PsPrioritySeparation的值。窗口子系统负责确定哪一个进程是前台进程。正如有关时间配额的讨论中的描述，变量PsPrioritySeparation是时间配额表的索引，用于选择前台进程中线程的时间配额。

这种针对前台线程的优先级提升是为了改进交互性应用的响应时间特征。在前台应用完成它的等待操作时小幅提升它的优先级，以使它更有可能马上进入运行状态。特别是后台有多个优先级相同的进程时，这种做法可有效改进前台应用的响应时间特征。

与其他类型的优先级提升不同，这种做法在Windows 2000专业版和Windows 2000服务器版中都有效；而且用户不能禁止这种优先级提升，甚至是在用户已利用Win32的函数SetThreadPriorityBoost禁止了其他的优先级提升策略时，也是如此。

4. 图形用户接口线程被唤醒后的优先级提升

拥有窗口的线程在被窗口活动唤醒(如收到窗口消息)时将得到一个幅度为2的额外优先级提升。窗口系统(Win32k.sys)在调用函数KeSetEvent时实施这种优先级提升，KeSetEvent函数调用设置一个事件，用于唤醒一个图形用户接口线程。与前一种线程优先级提升类似，这种优先级提升的原因是改进交互应用的响应时间。

5. 对处理器饥饿线程的优先级提升

想象如下情形：一个优先级为7的线程正处于运行状态，另一个优先级为4的线程在这种情况下是不会得到处理器使用权的。但如果一个优先级为11的线程正等待某种被优先级为4的线程锁定的资源，这个优先级为4的线程将永远得不到足够长的处理器时间来完成它的工作，并释放阻塞优先级为11的线程所需要的资源。Windows 2000/XP将如何处理这种情形？在有关虚拟存储的讨论中介绍了一个称为平衡集管理器(balance set manager)的用于内存管理的系统线程，它会每秒钟检查一次就绪队列，看一看是否存在一直在就绪队列中排队超过300个时钟中断间隔的线程。依据系统的时钟中断间隔的不同，300个时钟中断间隔的时间大约为3到4秒。如果找到这个线程，平衡集管理器将把该线程的优先级提升到15，并分配给它一个长度为正常值两倍的时间配额；当被提升线程用完它的时间配额后，该线程的优先级立即衰减到它原来的基本优先级。如果在该线程结束前出现其他高优先级的就绪线程，该线程会被放回就绪队列，并在就绪队列中超过另外300个时钟中断间隔后再次被提升优先级。

在每次运行时,平衡集管理器并不真正扫描所有就绪线程。为了减少它的处理器占用时间,平衡集管理器只扫描16个就绪线程。如果就绪队列中有更多的线程,它将记住暂停时的位置,并在下一次开始时从当前位置开始扫描。与此同时,平衡集管理器在每次扫描时最多提升10个线程的优先级。如果在一次扫描中已提升了10个线程的优先级(这表明系统处于特别繁忙的状态),平衡集管理器会停止本次扫描,并在下一次开始时从当前位置开始扫描。

这种算法并不能解决所有优先级倒置的问题,但它很有效。处于处理器饥饿的线程通常都能得到足够的处理器时间来完成它处理的操作并重新进入等待状态。

3.9.8 对称多处理器系统上的线程调度

如果完全基于线程优先级进行线程调度,在多处理器系统中会出现什么情况?当Windows 2000/XP试图调度优先级最高的可执行线程时,有几个因素会影响到处理器的选择。Windows 2000/XP只保证一个优先级最高的线程处于运行状态。在描述算法前,我们首先定义几个术语。

1. 亲合关系

每个线程都有一个亲合掩码,描述该线程可在哪些处理器上运行。线程的亲合掩码是从进程的亲合掩码继承得到的。缺省时,所有进程(即所有线程)的亲合掩码为系统中所有可用处理器的集合。也就是说,所有线程可在所有处理器上运行。应用程序通过调用SetProcessAffinityMask或SetThreadAffinityMask函数来修改缺省的亲合掩码。

2. 线程的首选处理器和第二处理器

每个线程在对应的内核线程控制块中都保存有两个处理器标识:

- 1) 首选处理器:线程运行时的偏好处理器。
- 2) 第二处理器:线程第二个选择的运行处理器。

线程的首选处理器是基于进程控制块的索引值在线程创建时随机选择的。索引值在每个线程创建时递增,这样进程中每个新线程得到的首选处理器会在系统中的可用处理器中循环。线程创建后,Windows 2000/XP系统不会修改线程的首选处理器设置;但应用程序可通过SetThreadIdealProcessor函数来修改线程的首选处理器。

3. 就绪线程的运行处理器选择

当线程进入运行状态时,Windows 2000/XP首先试图调度该线程到一个空闲处理器上运行。如果有多个空闲处理器,线程调度器的调度顺序为:首先是线程的首选处理器,其次是线程的第二处理器,第三是当前执行处理器(即正在执行调度器代码的处理器)。如果这些处理器都不是空闲的,Windows 2000/XP将依据处理器标识从高到低扫描系统中的空闲处理器状态,选择找到的第一个空闲处理器。

如果线程进入就绪状态时所有处理器都处于繁忙状态,Windows 2000/XP将检查它是否可抢先一个处于运行状态或备用状态的线程。检查的顺序如下:首先是线程的首选处理器,其次是线程的第二处理器。如果这两个处理器都不在线程的亲合掩码中,Windows 2000/XP将依据活动处理器掩码选择该线程可运行的编号最大的处理器。注:线程的亲合掩码与首选处理器、第二处理器的设置是相互独立的,首选和第二处理器由系统在创建线程时指定,而亲合掩码由用户选择。

如果被选中的处理器已有一个线程处于备用状态(即下一个在该处理器上运行的线程),并且该线程的优先级低于正在检查的线程,则正在检查的线程取代原处于备用状态的线程,成为该处理器的下一个运行线程。如果已有一个线程正在被选中的处理器上运行,Windows 2000/XP将检查当前运行线程的优先级是否低于正在检查的线程;如果正在检查的线程优先级高,则标记当前运行线程为被抢先,系统会发出一个处理器间中断,以抢先正在运行的线程,让新线程在该处理器上运行。

注:Windows 2000/XP并不检查所有处理器上的运行线程和备用线程的优先级,而仅仅按上述过程检查一个被选中处理器上的运行线程和备用线程的优先级。如果在被选中的处理器上没有线程可被抢先,则新线程放入相应优先级的就绪队列,并等待调度执行。

4. 为特定的处理器调度线程

在有些情况(如线程降低它的优先级、修改它的亲合处理器、推迟或放弃执行等)下,Windows 2000/XP必须选择一个新线程,在刚让出的处理器上运行。在单处理器系统中,Windows 2000/XP简单地从就绪队列中选择最高优先级的第一个线程。在多处理器系统,Windows 2000/XP不能简单地从就绪队列中取第一个线程,它要寻找一个满足下列四个条件之一的线程。

- 1) 线程的上一次运行是在该处理器上。
- 2) 线程的首选处理器是该处理器。
- 3) 处于就绪状态的时间超过2个时间配额单位。
- 4) 优先级大于等于24。

显然,由线程亲合掩码限制不能在指定处理器上运行的线程将在检查中跳过。如果Windows 2000/XP不能找到满足要求的线程,它将从就绪队列的队首取第一个线程进入运行状态。

为什么在为处理器选择运行线程时要考虑线程上一次运行时使用的处理器?主要的原因是速度问题。如果线程的连续两次运行都在同一个处理器上,将增加线程数据仍保留在处理器第二级缓存的可能性。

5. 最高优先级就绪线程可能不处于运行状态

如前所述,在多处理器系统中,Windows 2000/XP并不总是选择优先级最高的线程在一个指定的处理器上运行。于是有可能出现这种情况,一个比当前正在运行的线程优先级更高的线程处于就绪状态,但不能立即抢先当前线程,进入运行状态。

一种高优先级线程可能不抢先当前线程的情况是,线程的亲合掩码限制线程只能在一部分可用处理器上运行。在这种情况下,某线程可运行的处理器上运行着高优先级线程,而其他处理器可能空闲或运行着该线程可抢先的低优先级线程。虽然把一个处于运行状态的线程从一个处理器移到另一个处理器的做法可允许另一个由于亲合掩码限制而不能运行的线程进入运行状态,但Windows 2000/XP不会因此把一个正在运行的线程从一个处理器移到另一个处理器上。

例如,假设0号处理器上正运行着一个可在任何处理器上运行的优先级为8的线程,1号处理器上正运行着一个可在任何处理器上运行的优先级为4的线程;这时一个只能在0号处理器上运行的优先级为6的线程进入就绪状态。在这种情况下,优先级为6的线程只能等待0号处理器上优先

级为8的线程结束。因为Windows 2000/XP不会为了让优先级为6的线程在0号处理器上运行，而把优先级为8的线程从0号处理器移到1号处理器。即0号处理器上的优先级为8的线程不会抢先1号处理器上优先级为4的线程。

3.9.9 空闲线程

如果在一个处理器上没有可运行的线程，Windows 2000/XP会调度相应处理器上对应的空闲线程。因为在多处理器系统中可能两个处理器同时运行空闲线程，所以系统中的每个处理器都有一个对应的空闲线程。Windows 2000/XP给空闲线程指定的线程优先级为0，但实际上该空闲线程只在没有其他线程要运行时才运行。空闲线程的功能就是在一个循环中检测是否有要进行的工作。虽然不同处理器结构下空闲线程的流程有一些区别，但基本的控制流程都是如下所描述的过程。

- 1) 处理所有待处理的中断请求。
- 2) 检查是否有待处理的DPC请求。如果有，则清除相应软中断并执行DPC。
- 3) 检查是否有就绪线程可进入运行状态。如果有，调度相应线程进入运行状态。
- 4) 调用硬件抽象层的处理器空闲例程，执行相应的电源管理功能。

在Windows 2000/XP下有多种进程浏览工具，不同浏览工具给出的空闲线程的名字会不同，如系统空闲进程、空闲进程、系统进程等。

习题

- 3.1 试说明进程与程序的关系。
- 3.2 试比较进程与线程的区别。
- 3.3 在进程模型中引入挂起状态的目的是什么？
- 3.4 试分析内核线程、用户线程与轻量级进程的不同点。
- 3.5 试说明进程模型中等待状态与阻塞状态的区别。
- 3.6 写一个程序，通过递归创建进程1000次来测定Windows 2000中的进程创建速度。
- 3.7 写一个程序，通过递归创建线程1000次来测定Windows 2000中的线程创建速度。
- 3.8 写一个程序，通过创建新进程来执行一个命令列表。
- 3.9 写一个程序，通过创建新线程来执行一个命令列表。
- 3.10 写一个程序，每过2秒钟显示一行提示，一共显示10行提示信息。要求在前5行提示时，禁止用Ctrl-C来终止程序的执行；而在后5行提示时，可用Ctrl-C来终止程序的执行。
- 3.11 完成程序A，实现在创建后等待定长时间k后终止，k为命令行参数。完成程序B，在后台创建n个进程，执行程序A(命令行参数为1至2k间的一个随机数)；随后等待定长时间k后结束。观察整个过程中程序B和它所创建的n个进程的关系。
- 3.12 分别使用消息、管道机制来实现三个进程间的数据传递。第一个进程从数据文件中读入数据，并按字节进行变换T后传递给第二个进程。第二进程对收到的数据按字节进行变换T后传递给第三个进程。第三个进程对收到的数据按字节进行变换T后，把结果存入文件。变换T可为加1后模256。

- 3.13 试说明Windows 2000中的中断优先级与线程优先级的关系。
- 3.14 试说明Windows 2000中的线程优先级控制机制。
- 3.15 试说明在什么情况下线程的时间配额会被增加。
- 3.16 Windows 2000专业版和服务器的缺省时间配额是如何得到的？
- 3.17 试说明Windows 2000的线程调度算法是哪些处理器调度算法的综合。
- 3.18 为什么在Windows 2000中完成I/O操作后的线程优先级提升是以线程的基本优先级为基准点，而前台线程在等待结束后的优先级提升是以线程的当前优先级为基准点？
- 3.19 试说明Windows 2000在多处理器系统中的线程调度过程。
- 3.20 为什么Windows 2000在多处理器系统中不能保证各处理器上正在运行线程的优先级都不低于就绪线程的最高优先级？

参考文献

- 1. David A. Solomon, Mark E. Russinovich. Inside Microsoft Windows 2000. 3rd Edition. Microsoft Press, 2000
- 2. Gary Nutt. Operating System Project Using Windows NT. Addison-Wesley
- 3. The July 2000 release of the MSDN Library
- 4. William Stallings. Operating Systems. 3rd edition. 北京：清华大学出版社, 1998
- 5. 张尧学, 史美林编. 计算机操作系统教程. 北京：清华大学出版社, 1993
- 6. Uresh Vahalia. UNIX高级教程——系统技术内幕. 北京：清华大学出版社, 1999

第 4 章

存储体系

第 ④ 章

存储体系

存储器在计算机系统中起着非常重要的作用，它负责整个计算机系统中数据的保存。随着计算机技术的发展，计算机的体系结构已从以运算器为中心演变为以存储器为中心。存储器相关技术也在朝着高速度、大容量、小体积的方向飞速发展。随着制造工艺的不断提高，集成电路的集成度在不断地增大，内存的容量亦随之增大，内存的访问速度也有所提高，而单位容量所占的体积却在不断减小。作为辅存的磁介质存储器，由于磁头读写技术的不断进步，存储密度和访问速度也在大幅提高。DRAM的集成度每年约增加60%，每三年翻两番；磁介质存储器的存储密度，每年增加约50%多，也接近每三年翻两番。

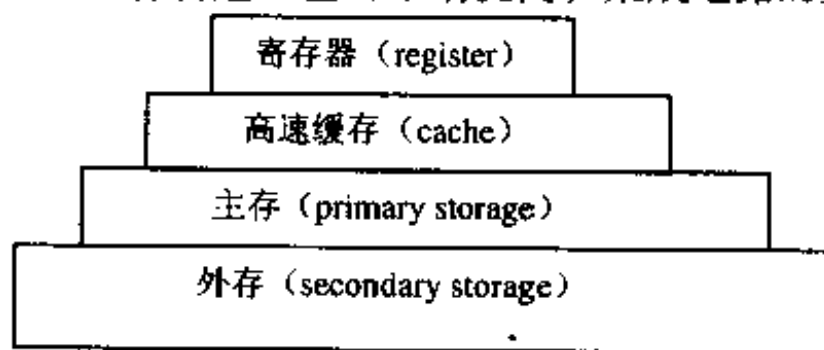


图4-1 存储层次结构

存储组织就是要在存储技术和CPU寻址技术许可的范围内寻求合理的存储结构，其依据是访问速度、匹配关系、容量要求和价格。常见的两种存储的组织形式是“寄存器-内存-外存”结构和“寄存器-缓存-内存-外存”结构。现在微机中的存储层次组织如图4-1所示：从上到下访问速度越来越慢，容量越来越大，价格越来越便宜。这种组织形式的最佳状态应是各层次的存储器都处于均衡的繁忙状态。

虽然，存储器的容量不断扩大、速度不断提高，但是仍然不能满足现代软件发展的需求，因此存储器仍然是一种宝贵的资源。如何对它们进行有效的管理，不仅直接影响到存储器的利用率，而且还对系统的性能有重大影响，因此存储管理是操作系统的一项非常重要的任务，也是不可缺少的部分。本章介绍了存储管理的基本原理，以及Windows 2000/XP的内存管理、外存管理和高速缓存管理三部分。

4.1 存储管理的基本原理

4.1.1 内存管理方法

内存管理主要包括内存分配和回收、地址变换、内存扩充、内存共享和保护等功能。下面主要介绍连续分配存储管理、覆盖与交换技术以及页式与段式存储管理等基本概念和原理。

1. 连续分配存储管理方式

连续分配是指为一个用户程序分配连续的内存空间。连续分配有单一连续存储管理和分区式

存储管理两种方式。

(1) 单一连续存储管理

在这种管理方式中，内存被分为两个区域：系统区和用户区。应用程序装入到用户区，可使用用户区全部空间。其特点是，最简单，适用于单用户、单任务的操作系统。CP/M和DOS 2.0以下就是采用此种方式。这种方式的最大优点就是易于管理。但也存在着一些问题和不足之处，例如：对要求内存空间少的程序，造成内存浪费；程序全部装入，使得很少使用的程序部分也占用一定数量的内存。

(2) 分区式存储管理

为了支持多道程序系统和分时系统，支持多个程序并发执行，引入了分区式存储管理。分区式存储管理是把内存分为一些大小相等或不等的分区，操作系统占用其中一个分区，其余的分区由应用程序使用，每个应用程序占用一个或几个分区。分区式存储管理虽然可以支持并发，但难以进行内存分区的共享。

分区式存储管理引入了两个新的问题：内碎片和外碎片。前者是占用分区内未被利用的空间，后者是占用分区之间难以利用的空闲分区（通常是小区间空闲分区）。为实现分区式存储管理，操作系统应维护的数据结构为分区表或分区链表。表中各表项一般包括每个分区的起始地址、大小及状态（是否已分配）。

分区式存储管理常采用的一项技术就是内存紧缩(compact)：将各个占用分区向内存一端移动，然后将各个空闲分区合并成为一个空闲分区。这种技术在提供了某种程度上的灵活性的同时，也存在着一些弊端，例如：对占用分区进行内存数据搬移占用CPU时间；如果对占用分区中的程序进行“浮动”，则其重定位需要硬件支持。

1) 固定分区(fixed partitioning)。固定式分区的特点是把内存划分为若干个固定大小的连续分区。分区大小可以相等：这种作法只适合于多个相同程序的并发执行（处理多个类型相同的对象）。分区大小也可以不等：有多个小区间、适量的中等分区以及少量的大分区。根据程序的大小，分配当前空闲的、适当大小的分区。这种技术的优点在于，易于实现，开销小。缺点主要有两个：内碎片造成浪费；分区总数固定，限制了并发执行的程序数目。

2) 动态分区(dynamic partitioning)。动态分区的特点是动态创建分区：在装入程序时按其初始要求分配，或在其执行过程中通过系统调用进行分配或改变分区大小。与固定分区相比较其优点是：没有内碎片。但它却引入了另一种碎片——外碎片。动态分区的分区分配就是寻找某个空闲分区，其大小需大于或等于程序的要求。若是大于要求，则将该分区分割成两个分区，其中一个分区为要求的大小并标记为“占用”，而另一个分区为余下部分并标记为“空闲”。分区分配的先后次序通常是从内存低端到高端。动态分区的分区释放过程中有一个要注意的问题是，将相邻的空闲分区合并成一个大的空闲分区。

下面列出了几种常用的分区分配算法：

- 首先适配法(first-fit)：按分区在内存的先后次序从头查找，找到符合要求的第一个分区进行分配。该算法的分配和释放的时间性能较好，较大的空闲分区可以被保留在内存高端。但随着低端分区不断划分会产生较多小区间，每次分配时查找时间开销便会增大。

- 下次适配法(next-fit): 按分区在内存的先后次序, 从上次分配的分区起查找(到最后分区时再从头开始), 找到符合要求的第一个分区进行分配。该算法的分配和释放的时间性能较好, 使空闲分区分布得更均匀, 但较大空闲分区不易保留。
- 最佳适配法(best-fit): 按分区在内存的先后次序从头查找, 找到其大小与要求相差最小的空闲分区进行分配。从个别来看, 外碎片较小; 但从整体来看, 会形成较多外碎片。优点是较大的空闲分区可以被保留。
- 最坏适配法(worst-fit): 按分区在内存的先后次序从头查找, 找到最大的空闲分区进行分配。基本不留下小空闲分区, 不易形成外碎片。但由于较大的空闲分区不被保留, 当有对内存需求较大的进程需要运行时, 其要求不易被满足。

2. 覆盖和交换技术

引入覆盖(overlay)技术的目标是在较小的可用内存中运行较大的程序。这种技术常用于多道程序系统之中, 与分区式存储管理配合使用。覆盖技术的原理很简单, 一个程序的几个代码段或数据段, 按照时间先后来占用公共的内存空间。将程序必要部分(常用功能)的代码和数据常驻内存; 可选部分(不常用功能)平时存放在外存(覆盖文件)中, 在需要时才装入内存。不存在调用关系的模块不必同时装入到内存, 从而可以相互覆盖。覆盖技术的缺点是编程时必须划分程序模块和确定程序模块之间的覆盖关系, 增加编程复杂度; 从外存装入覆盖文件, 以时间延长来换取空间节省。覆盖的实现方式有两种: 以函数库方式实现或操作系统支持。

交换(swapping)技术在多个程序并发执行时, 可以将暂时不能执行的程序送到外存中, 从而获得空闲内存空间来装入新程序, 或读入保存在外存中而处于就绪状态的程序。交换单位为整个进程的地址空间。交换技术常用于多道程序系统或小型分时系统中, 与分区式存储管理配合使用, 又称作“对换”或“滚进/滚出”(roll-in/roll-out)。其优点之一是增加并发运行的程序数目, 并且给用户适当的响应时间; 与覆盖技术相比交换技术另一个显著的优点是不影响程序结构。交换技术本身也存在着不足, 例如: 对换入和换出的控制增加处理器开销; 程序整个地址空间都进行对换, 没有考虑执行过程中地址访问的统计特性。

3. 页式和段式存储管理

在前面的几种存储管理方法中, 为进程分配的空间是连续的, 使用的地址都是物理地址。如果允许将一个进程分散到许多不连续的空间, 就可以避免内存紧缩, 减少碎片。基于这一思想, 通过引入进程的逻辑地址, 把进程地址空间与实际存储空间分离, 增加存储管理的灵活性。地址空间和存储空间两个基本概念的定义如下:

- 地址空间: 将源程序经过编译后得到的目标程序, 存在于它所限定的地址范围内, 这个范围称为地址空间。地址空间是逻辑地址的集合。
- 存储空间: 指主存中一系列存储信息的物理单元的集合, 这些单元的编号称为物理地址。存储空间是物理地址的集合。

根据分配时所采用的基本单位不同, 可将离散分配的管理方式分为以下三种: 页式存储管理、段式存储管理和段页式存储管理。其中段页式存储管理是前两种结合的产物。

(1) 页式存储管理

1) 基本原理。将程序的逻辑地址空间划分为固定大小的页(page)，而物理内存划分为同样大小的页框(page frame)。程序加载时，可将任意一页放入内存中任意一个页框，这些页框不必连续，从而实现了离散分配。该方法需要CPU的硬件支持，来实现逻辑地址和物理地址之间的映射。在页式存储管理方式中地址结构由两部分组成，前一部分是页号，后一部分为页内地址，如图4-2所示。

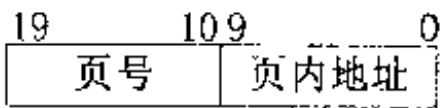


图4-2 页的划分

这种管理方式的优点是，没有外碎片，每个内碎片不超过页大小。比前面所讨论的几种管理方式的最大进步是，一个程序不必连续存放。这样就便于改变程序占用空间的大小（主要指随着程序运行，动态生成的数据增多，所要求的地址空间相应增长）。缺点是仍旧要求程序全部装入内存，没有足够的内存，程序就不能执行。

2) 页式管理的数据结构。在页式系统中进程建立时，操作系统为进程中所有的页分配页框。当进程撤销时收回所有分配给它的页框。在程序的运行期间，如果允许进程动态地申请空间，操作系统还要为进程申请的空间分配物理页框。操作系统为了完成这些功能，必须记录系统内存中实际的页框使用情况。操作系统还要在进程切换时，正确地切换两个不同的进程地址空间到物理内存空间的映射。这就要求操作系统要记录每个进程页表的相关信息。为了完成上述的功能，一个页式系统中，一般要采用如下的数据结构。

- 进程页表：完成逻辑页号（本进程的地址空间）到物理页面号（实际内存空间）的映射。每个进程有一个页表，描述该进程占用的物理页面及逻辑排列顺序。
- 物理页面表：整个系统有一个物理页面表，描述物理内存空间的分配使用状况，其数据结构可采用位示图和空闲页链表。
- 请求表：整个系统有一个请求表，描述系统内各个进程页表的位置和大小，用于地址转换，也可以结合到各进程的PCB（进程控制块）里。

3) 页式管理的地址变换。在页式系统中，指令所给出的地址分为两部分：逻辑页号和页内地址。CPU中的内存管理单元（MMU）按逻辑页号通过查进程页表得到物理页框号，将物理页框号与页内地址相加形成物理地址（见图4-3）上述过程通常由处理器的硬件直接完成，不需要软件参与。通常，操作系统只需在进程切换时，把进程页表的首地址装入处理器特定的寄存器中即可。一般来说，页表存储在主存之中。这样处理器每访问一个在内存中的操作数，就要访问两次内存。第一次用来查找页表将操作数的逻辑地址变换为物理地址；第二次完成真正的读写操作。这样做时间上耗费严重。为缩短查找时间，可以将页表从内存装入CPU内部的关联存储器（例如，快表）中，实现按内容查找。此时的地址变换过程是：在CPU给出有效地址后，由地址变换机构自动将页号送入快表，并将此页号与快表中的所有页号进行比较，而且这种比较是同时进行的。若其中有与此相匹配的页号，表示要访问的页的页表项在快表中。于是可直接读出该页所对应的物理页号，这样就无需访问内存中的页表。由于关联存储器的访问速

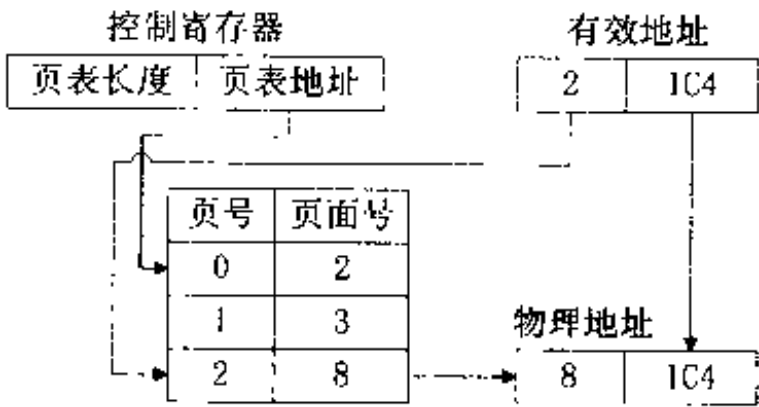


图4-3 页式地址变换

度比内存的访问速度快得多，这样就可以大大减少访问内存的时间。仅在快表中没有相应的页表项时，才访问内存中的页表。如果快表的命中率较高，时间上的牺牲就相对较小，而对存储管理的性能就会有很大提高。

(2) 段式存储管理

1) 基本原理。在段式存储管理中，将程序的地址空间划分为若干个段(segment)，这样每个进程有一个二维的地址空间。在前面所介绍的动态分区分配方式中，系统为整个进程分配一个连续的内存空间。而在段式存储管理系统中，则为每个段分配一个连续的分区，而进程中的各个段可以不连续地存放在内存的不同分区中。程序加载时，操作系统为所有段分配其所需内存，这些段不必连续，物理内存的管理采用动态分区的管理方法。在为某个段分配物理内存时，可以采用首先适配法、下次适配法、最佳适配法等方法。在回收某个段所占用的空间时，要注意将收回的空间与其相邻的空间合并。段式存储管理也需要硬件支持，实现逻辑地址到物理地址的映射。程序通过分段划分为多个模块，如代码段、数据段、共享段。这样做的优点是：可以分别编写和编译源程序的一个文件，并且可以针对不同类型的段采取不同的保护，也可以按段为单位来进行共享。总的来说，段式存储管理的优点是：没有内碎片，外碎片可以通过内存紧缩来消除；便于实现内存共享。缺点与页式存储管理的缺点相同，进程必须全部装入内存。

2) 段式管理的数据结构。为了实现段式管理，操作系统需要如下的数据结构来实现进程的地址空间到物理内存空间的映射，并跟踪物理内存的使用情况，以便在装入新的段的时候，合理地分配内存空间。

- 进程段表：描述组成进程地址空间的各段，可以是指向系统段表中表项的索引。每段有段基址(base address)。
- 系统段表：系统所有占用段。
- 空闲段表：内存中所有空闲段，可以结合到系统段表中。

3) 段式管理的地址变换。在段式管理系统中，整个进程的地址空间是二维的，即其逻辑地址由段号和段内地址两部分组成。为了完成进程逻辑地址到物理地址的映射，处理器会查找内存中的段表，由段号得到段的首地址，加上段内地址，得到实际的物理地址（见图4-4）。这个过程也是由处理器的硬件直接完成的，操作系统只需在进程切换时，将进程段表的首地址装入处理器的特定寄存器当中。这个寄存器一般被称作段表地址寄存器。

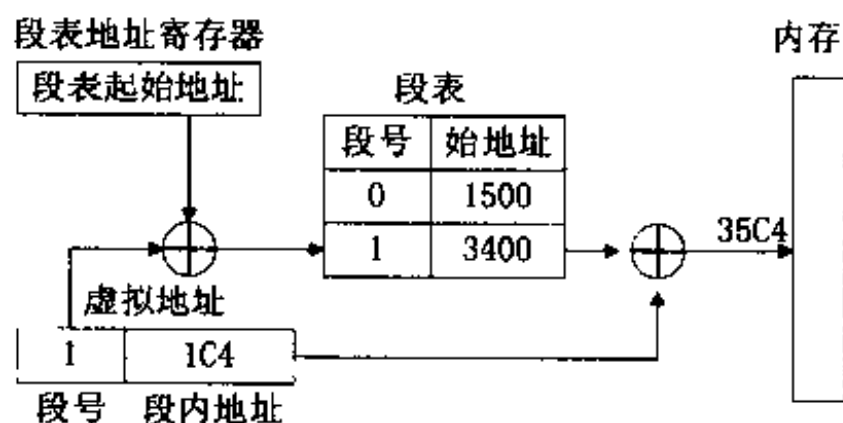


图4-4 段式地址变换

与页式系统类似，段表由于放在内存中，每访问一次数据都要两次访问内存，从而降低了计算机的速度。解决这个问题的方法，也与解决页式系统中相同问题的方法类似，在处理器中增设一个高速关联存储器，用于保存经常使用的段表项。由于一般段表项的数目比页表项的数目少得多，其所需的关联存储器也相对较小，因此可以显著减少存取数据的时间。

4. 页式和段式系统的区别

页式和段式系统有许多相似之处。比如，两者都采用离散分配方式，且都通过地址映射机构来实现地址变换。但概念上两者也有很多区别，主要表现在：

- 页是信息的物理单位，分页是为了实现离散分配方式，以减少内存的外零头，提高内存的利用率。或者说，分页仅仅是由于系统管理的需要，而不是用户的需要。段是信息的逻辑单位，它含有一组其意义相对完整的信息。分段的目的是为了更好地了解用户的需要。
- 页的大小固定且由系统决定，把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的。段的长度不固定，且决定于用户所编写的程序，通常由编译系统在对源程序进行编译时根据信息的性质来划分。
- 页式系统地址空间是一维的，即单一的线性地址空间，程序员只需利用一个标识符，即可表示一个地址。分段的作业地址空间是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址。

4.1.2 虚拟存储器

1. 局部性原理

局部性原理(principle of locality)指程序在执行过程中的一个较短时期，所执行的指令地址和指令操作数的地址，分别局限于一定区域内。它可以表现为：时间局部性，即一条指令的一次执行和下次执行，一个数据的一次访问和下次访问，都集中在一个较短时期内；空间局部性，即当前指令和邻近的几条指令，当前访问的数据和邻近的数据，都集中在一个较小区域内。

局部性原理的产生原因可以归纳为如下几种情况：

- 程序在执行时，大部分是顺序执行的指令，少部分是转移和过程调用指令。
- 过程调用的嵌套深度一般不超过5，因此执行的范围不超过这组嵌套的过程。
- 程序中存在相当多的循环结构，它们由少量指令组成却被多次执行。
- 程序中相当多对一定数据结构的操作，如数组操作，往往局限在较小范围内。

2. 虚拟存储器的基本原理

基于局部性原理，在程序装入时，不必将其全部读入到内存，而只需将当前需要执行的部分页或段读入内存，就可让程序开始执行。在程序执行过程中，如果需执行的指令或访问的数据尚未在内存（称为缺页或缺段），则由处理器通知操作系统将相应的页或段调入到内存，然后继续执行程序。另一方面，操作系统将内存中暂时不使用的页或段调出，保存在外存上，从而腾出空间存放将要装入的程序以及将要调入的页或段。从用户的角度看，该系统具有的内存容量，将比实际的内存容量大得多，所以称之为虚拟存储器。

(1) 引入虚拟存储技术的好处

虚拟存储技术的引入，提供给用户一个大于实际物理内存的虚拟存储空间。这使在较小的可用内存中执行较大的用户程序成为可能，并且可在内存中容纳更多程序的并发执行。与覆盖技术比较，虚拟存储技术的一个显著优点是不影响编程时的程序结构，也就是说虚拟存储器对程序员是透明的。

(2) 虚拟存储技术的种类

虚拟存储技术分为三类：请求分页、请求分段和请求段页式。请求分页（段）存储管理在页式（段式）存储管理的基础上，增加了请求调页（调段）等功能。请求段页式存储管理是请求分

页和请求分段存储管理的结合。段页式存储管理的分配单位是段和页。在段页式存储管理中，逻辑地址是由段号、页号和页内偏移地址三部分组成的。地址变换的过程也分为两步，先查段表，再查该段的页表。在地址变换的过程中会产生缺段中断和缺页中断两种不同类型的中断。

(3) 虚拟存储技术的特征

虚拟存储技术的基本特征是物理内存分配不连续，虚拟地址空间使用也不连续，如数据段和栈段之间可以存在一定的空闲虚地址空间。与交换技术不同的是调入和调出是对部分虚拟地址空间进行的，通过物理内存和外存相结合，可提供大范围的虚拟地址空间。

3. 请求分页系统

(1) 请求分页对页表的扩充

在请求分页系统中所使用的主要数据结构仍然是页表。它对页式系统中的页表机制进行了扩充。但其基本作用是实现由用户地址空间到物理内存空间的映射。由于只将应用程序的一部分装入内存，还有一部分仍在磁盘上，故需在页表中增加若干项，供操作系统实现虚拟存储器功能时参考。

常见的系统中，一般对页表的表项进行如下扩充：除了页号对应的物理块号，还增加了状态位、修改位、外存地址和访问字段等。

- 状态位，用于指示该页是否已经调入了内存。该位一般由操作系统软件来管理，每当操作系统把一页调入物理内存中时，置位。相反，当操作系统把该页从物理内存调出时，复位。CPU对内存进行引用时，根据该位判断要访问的页是否在内存中，若不在内存之中，则产生缺页中断。
- 修改位，表示该页调入内存后是否被修改过。当CPU以写的方式访问页面时，对该页的页表项中的修改位置位。该位也可由操作系统软件来修改，例如，当操作系统将修改过的页面保存在磁盘上后，可将该位复位。
- 外存地址，用于指出该页在外存上的地址，供调入该页时使用。
- 访问字段，用于记录本页在一定时间内被访问的次数，或最近已经有多长时间未被访问。提供给相应的置换算法在选择换出页面时参考。

(2) 对缺页中断的支持

在请求分页系统中，CPU硬件一定要提供对缺页中断的支持，根据页表项中的状态位判断是否产生缺页中断。缺页中断是一个比较特殊的中断，这主要体现在如下两点：

- 在指令的执行期间产生和处理缺页信号。通常的CPU外部中断，是在每条指令执行完毕后检查是否有中断请求到达。而缺页中断，是在一条指令的执行期间，发现要访问的指令和数据不在内存时产生和处理的。
- 一条指令可以产生多个缺页中断。例如，一条双操作数的指令，每个操作数都不在内存中，这条指令执行时，将产生两个中断。CPU提供的硬件支持，还要体现在当从中断处理程序返回时，能够正确执行产生缺页中断的指令。

(3) 页面调度策略

虚拟存储器系统通常定义三种策略来规定如何（或何时）进行页面调度：调入策略、置页策略和置换策略。

1) 页面调入策略

虚拟存储器的调入策略决定了什么时候将一个页由外存调入内存之中。在虚拟页式管理中有两种常用调入策略：

- 请求调页(demand paging)：只调入发生缺页时所需的页面。这种调入策略实现简单，但容易产生较多的缺页中断，造成对外存I/O次数多，时间开销过大，容易产生抖动现象。
- 预调页(prepaging)：在发生缺页需要调入某页时，一次调入该页以及相邻的几个页。这种策略提高了调页的I/O效率，减少了I/O次数。但由于这是一种基于局部性原理的预测，若调入的页在以后很少被访问，则造成浪费。这种方式常在程序装入时使用。

通常对外存交换区的I/O效率比文件区的高。关于调入页面来源，通常有两种做法：

- 进程装入时，将其全部页面复制到交换区，以后总是从交换区调入。执行时调入速度快，要求交换区空间较大。
- 凡是未被修改的页，都直接从文件区读入，而被置换时不需调出；已被修改的页面，被置换时需调出到交换区，以后从交换区调入。这种方式节省了交换区空间，但也可能引发某些问题。如在Solaris 2.3中，装入可执行文件a从而创建进程P，如果在P执行时，改写了可执行文件a（如重新编译和连接，不包括用touch命令使文件的最后修改时间变得更晚），而此后P发生缺页需要从a中调入页面，则可能会因为各个页面内容无法配合而出错（如Bus Error或 Segmentation Fault）。

2) 置页策略 当线程产生缺页中断时，内存管理器还必须确定将调入的虚拟页放在物理内存的何处。用于确定最佳位置的一组规则称为“置页策略”。选择页框应使CPU内存高速缓存不必要的震荡最小，因此Windows 2000/XP 需要考虑CPU内存高速缓存的大小。

3) 置换策略 如果缺页中断发生时物理内存已满，“置换策略”被用于确定哪个虚页面必须从内存中移出，为新的页面腾出空位。在请求分页系统中，可采用两种分配策略，即固定分配和可变分配。在进行置换时，也可以采用两种策略，即全局置换和局部置换。将它们组合起来，有如下三种策略。

- 固定分配局部置换 (fixed allocation, local replacement)。可基于进程的类型，为每一进程分配固定页数的内存空间，在整个运行期间不再改变。采用该策略时，如果进程在运行中出现缺页，则只能从该进程的N个页面中选择一个换出，然后再调入一页，以保证分配给该进程的内存空间不变。
- 可变分配全局置换 (variable allocation, global replacement)。采用这种策略时，先为系统中的每一进程分配一定数量的物理块，操作系统本身也保持一个空闲物理块队列。当某进程发生缺页时，由系统的空闲物理块队列中取出--物理块分配给该进程。当空闲物理块队列中的物理块用完时，操作系统从内存中选择一块调出。该块可能是系统中任意一个进程的页。
- 可变分配局部置换 (variable allocation, local replacement)。同样基于进程的类型，为每一进程分配一定数目的内存空间。但当某进程发生缺页时，只允许从该进程的页面中选出一页换出，这样就不影响其他进程的运行。如果进程在运行的过程中，频繁地发生缺页中断，则系统再为该进程分配若干物理块，直到进程的缺页率降低到适当程度为止。

(4) 置换算法

置换算法 (replacement algorithm) 决定在需要调入页面时, 选择内存中哪个物理页面被置换。置换算法的出发点应该是, 把未来不再使用的或短期内较少使用的页面调出。而未来的实际情况是不确定的, 通常只能在局部性原理指导下依据过去的统计数据进行分析。常用的算法有以下几种:

- 最佳算法(optimal, OPT)。选择“未来不再使用的”或“在离当前最远位置上出现的”页面被置换。这是一种理想情况, 是实际执行中无法预知的, 因而不能实现, 只能用作性能评价的依据。
- 最近最久未使用算法(Least Recently Used, LRU)。选择内存中最久未使用的页面被置换。这是局部性原理的合理近似, 性能接近最佳算法。但由于需要记录页面使用时间的先后关系, 硬件开销太大。LRU可用如下的硬件机构帮助实现:
 - 一个特殊的栈: 把被访问的页面移到栈顶, 于是栈底的是最久未使用页面。
 - 每个页面设立移位寄存器: 被访问时左边最高位置1, 定期右移并且最高位补0, 于是寄存器数值最小的是最久未使用页面。
- 先进先出算法(FIFO)。选择装入最早的页面置换。可以通过链表来表示各页的装入时间先后。FIFO的性能较差, 因为较早调入的页往往是经常被访问的页, 这些页在FIFO算法下被反复调入和调出, 并且有Belady现象。所谓Belady现象是指: 采用FIFO算法时, 如果对一个进程未分配它所要求的全部页面, 有时就会出现分配的页面数增多但缺页率反而提高的异常现象。Belady现象可形式化地描述为: 一个进程 P 要访问 M 个页, OS分配 N 个内存页面给进程 P ; 对一个访问序列 S , 发生缺页次数为 $PE(S, N)$ 。当 N 增大时, $PE(S, N)$ 时而增大, 时而减小。Belady现象的原因是FIFO算法的置换特征与进程访问内存的动态特征是矛盾的, 即被置换的页面并不是进程不会访问的。
- 时钟(clock)算法。也称最近未使用算法(Not Recently Used, NRU), 它是LRU和FIFO的折衷。每页有一个使用标志位(use bit), 若该页被访问则置user bit=1, 这是由CPU的硬件自动完成的。置换时采用一个指针, 从当前指针位置开始按地址先后检查各页, 寻找use bit=0的页面作为被置换页。指针经过的user bit=1的页都修改user bit=0, 这个修改的过程是操作系统完成的, 最后指针停留在被置换页的下一个页。
- 最不常用算法(Least Frequently Used, LFU)。选择到当前时间为止被访问次数最少的页面被置换。每页设置访问计数器, 每当页面被访问时, 该页面的访问计数器加1。发生缺页中断时, 淘汰计数值最小的页面, 并将所有计数清零。
- 页面缓冲算法(page buffering)。它是对FIFO算法的发展, 通过建立置换页面的缓冲, 这样就有机会找回刚被置换的页面, 从而减少系统I/O的开销。页面缓冲算法用FIFO算法选择被置换页, 把被置换的页面放入两个链表之一。即是如果页面未被修改, 就将其归入到空闲页面链表的末尾, 否则将其归入到已修改页面链表。空闲页面和已修改页面, 仍停留在内存中一段时间, 如果这些页面被再次访问, 只需较小开销, 被访问的页面就可以返还作为进程的内存页。需要调入新的物理页面时, 将新页面内容读入到空闲页面链表的第一项所指的页面, 然后将第一项删除。当已修改页面达到一定数目后, 再将它们一起调出到外存, 然后将它们归入空闲页面链表。这样能大大减少I/O操作的次数。

4. 工作集

工作集理论是在1968年由Denning提出并推广的。Denning认为程序在运行时对页面的访问是不均匀的：即往往在某段时间内的访问仅局限于较少的页面；而在另一段时间内，则又可能仅局限于对另一些较少的页面进行访问。如果能够预知程序在某段时间间隔内要访问哪些页面，并能提前将它们调入内存，将会大大降低缺页率，减少置换工作，提高CPU的利用率。

所谓工作集是指在某段时间间隔 Δ 里，进程实际要访问的页面集合。Denning认为，虽然程序只需少量的几页已在内存就可运行，但为使程序能有效地运行，较少地产生缺页，就必须使程序的工作集全部在内存中。然而，由于我们无法预知程序在不同时刻将访问哪些页面，因而只能像置换算法那样，利用程序过去某段时间内的行为，作为程序在将来某段时间内行为的近似。具体地说，便是把某进程在时间 t 的工作集记为 $w(t, \Delta)$ ，把变量 Δ 称为工作集窗口尺寸（windows size）。

正确选择工作集窗口的大小，对存储器的有效利用和系统吞吐率的提高，都将产生重要影响。如果把 Δ 选得很大，这种情况进程虽不易产生缺页，但存储器也将不会得到充分地利用。另一方面，如果把 Δ 选得过小，则会使进程在运行过程中频繁地产生缺页中断，反而降低了系统的吞吐量。

5. 请求分段系统

(1) 对段表的扩充

请求分段系统在段式存储管理的基础上，增加请求调段和段置换功能。为实现虚拟存储器的各项功能和管理需要在进程段表中添加若干项：

- 标志位：如存在位(present bit)，修改位(modified bit/dirty bit)，增长位（该段是否增长过）。
- 访问统计：如使用位(use bit)。
- 存取权限：如读R，写W，执行X。
- 外存地址：本段在外存的起始地址。

这些位和字段的意义与请求分页系统的页表项中的相应位和字段项的意义相似。

(2) 缺段中断

在请求分段系统中，采用的是请求调段策略。CPU硬件逻辑要根据段表表项进行地址变换或者产生缺段中断。请求分段存储管理与请求分页存储管理不同之处在于，指令和操作数必定不会跨越在段边界上。但是，在请求分段系统中，一般会增加存取权限的违例中断，和段的越界中断。

(3) 请求分段系统的内存分配策略

在请求分段系统中，对物理内存进行分配可采用与动态分区相似的最佳适配、首先适配等分配策略。

4.1.3 磁盘存储管理

磁盘存储器不仅容量大，存取速度快，而且可以实现随机存取，是实现虚拟存储器所必需的硬件。因此在现代计算机系统中，都配置了磁盘存储器，并以它为主，存放文件。磁盘存储管理的主要任务是：

- 为文件分配必要的存储空间；
- 提高磁盘存储空间的利用率；
- 提高对磁盘的I/O速度，以改善文件系统的性能；
- 采取必要的冗余措施，来确保文件系统的可靠性。

1. 磁盘性能简述

下面对磁盘性能，如对数据的组织、对磁盘的类型及访问时间等，作简要介绍。

(1) 数据的组织

磁盘设置中，可包含一个或多个盘片。每片分两面，每面又可分成若干条磁道（典型值为500~2000条磁道）。磁盘之间留有必要的空隙。为使处理简单起见，在每条磁道上可存储相同数目的二进制位。这样，磁盘密度即每英寸中所存储的位数。显然，内层磁道的密度较外层磁道的密度高。每条磁道又分成若干个扇区，其典型值为10~100个扇区。每个扇区的大小相当于一个盘块。各磁道之间同样要保留一定的间隙。

为了在磁盘上存储数据，必须将磁盘格式化。例如，一种温盘（温彻斯特盘）中一条磁道格式化后含有30个固定大小的扇区，每个扇区容量为600个字节。其中，512字节用于存放数据，其余用于存放控制信息。每个扇区包括两个字段：

- 标识符字段。其中一个字节作为该字段的定界符，利用磁道号、磁头号及扇区三者来标识一个扇区；CRC字段用于段校验。
- 数据字段。存放512个字节的数据。

(2) 磁盘的类型

对磁盘可从不同的角度进行分类。最常见的有：将磁盘分成硬盘和软盘，单片盘和多片盘。面定头磁盘和移动头磁盘等。下面仅对固定头磁盘和移动头磁盘作一介绍。

1) 面定头磁盘。这种磁盘在每条磁道上都有一个读/写磁头，所有的磁头都被装在一刚性磁臂中，通过这些磁头可访问所有的磁道，并进行并行读/写，有效地提高了磁盘的I/O速度。这种结构的磁盘主要用于大容量磁盘上。

2) 移动头磁盘。每一个盘面仅配有一个磁头，也被装入磁臂中，为能访问该盘面上的所有磁道，该磁头必须能移动以进行寻道。可是，移动头磁盘只能进行串行读/写，致使I/O速度较慢，但由于结构简单，故仍广泛地用于中、小型磁盘设备中。在微机上配置的磁盘，都采用移动磁头结构，故本节主要针对这类磁盘的I/O进行讨论。

(3) 磁盘访问时间

对磁盘的访问时间，包括以下三部分：

- 寻道时间 T_s 。这是把磁臂（磁头）从当前位置移动到指定磁道上所经历的时间。该时间是启动磁盘的时间 s 与磁头移动 n 条磁道所花费的时间之和。即

$$T_s = m \times n + s$$

式中， m 是一常数，它与磁盘驱动器的速度有关。对一般磁盘， $m = 0.3$ ；对高速磁盘， $m \leq 0.1$ 。磁盘启动时间约为3 ms。这样，对一般的温盘，其寻道时间将随寻道距离的增大而增大，大体上是10~40 ms。

- 旋转延迟时间 T_r 。 T_r 是指定扇区移动到磁头下面所经历的时间。对于硬盘，典型的旋转速度为3 600 r/min，每转需时16.7 ms，平均旋转延迟时间 T_r 为8.3 ms。对于软盘，其旋转速度为300或600 r/min，这样，平均 T_r 为50~100 ms。
- 传输时间 T_t 。 T_t 是指把数据从磁盘读出或向磁盘写入数据所经历的时间， T_t 的大小与每次所读/写的字节数 b 及旋转速度有关：

$$T_r = \frac{b}{rN}$$

式中, r 为磁盘以秒计的旋转速度; N 为一条磁道上的字节数。假设平均移动距离为半条磁道。因此, 可将访问时间 T_a 表示为

$$T_a = T_r + \frac{1}{2r} + \frac{b}{rN}$$

由上式可以看出, 在访问期间, 寻道时间和旋转延迟时间, 基本上都与所读/写数据的多少无关, 而且它通常是占据了访问时间的大头。例如, 我们假定寻道时间和旋转延迟时间平均为30 ms, 而磁道的传输速率为1 MB/s, 如果传输1K字节, 此时总的访问时间为31 ms, 传输时间所占比例是相当地小。当传输10K字节的数据时, 其访问时间也只是40 ms, 即当传输的数据量增加10倍时, 访问时间只增加了约30%。目前磁盘的传输速率已达20 MB/s以上, 数据传输时间所占的比例更低。可见, 适当地集中数据(不要太零散)传输, 将有利于提高传输效率。

2. 磁盘调度算法

磁盘是可被多个进程共享的设备。当有多个进程都请求访问磁盘时, 应采用一种适当的调度算法, 以使各进程对磁盘的平均访问(主要是寻道)时间最小。由于在访问磁盘的时间中, 主要是寻道时间, 因此, 磁盘调度的目标应是使磁盘的平均寻道时间最少。目前常用的磁盘调度算法有: 先来先服务; 最短寻道时间优先; 扫描算法; 循环扫描算法等。

(1) 先来先服务 (First-Come, First-Served, FCFS)

这是一种简单的磁盘调度算法。它根据进程请求访问磁盘的先后次序进行调度。此算法的优点是公平、简单, 且每个进程的请求都能依次得到处理, 不会出现某一进程的请求长期得不到满足的情况。但此算法由于未对寻道进行优化, 致使平均寻道时间可能较长。图4-5示出了有9个进程先后提出磁盘I/O请求时, 按FCFS算法进行调度的情况。这里, 将进程号(请求者)按其发出请求的先后次序排列。这样, 平均寻道距离为55.3条磁道。与后面要讲的几种调度算法相比, 其平均寻道距离较大。故FCFS算法仅适用于请求磁盘I/O的进程数目较少的场合。

(2) 最短寻道时间优先 (Shortest Seek Time First, SSTF)

该算法选择这样的进程, 其要求访问的磁道与当前磁头所在的磁道距离最近, 以使每次的寻道时间最短, 但这种调度算法却不能保证平均寻道时间最短。图4-6所示按SSTF算法进行调度时, 各进程被调度的次序, 每次磁头的移动距离, 以及9次磁头移动的平均距离。比较图4-5和图4-6可以看出, SSTF算法的平均每次磁头移动距离, 明显低于FCFS的距离。SSTF较之FCFS有更好的寻道性能, 故过去一度被广泛采用过。

(3) 各种扫描算法

1) 扫描(SCAN)算法。SSTF算法虽然获得较好的寻道性能, 但它可能导致某些进程发生“饥饿”(starvation)。因为只要不断有新进程到达, 且其访问的

(从100 ^号 磁道开始)	
被访问的下一个磁道号	移动距离 (磁道数)
55	45
58	3
39	19
18	21
90	72
160	70
150	10
38	112
184	146
平均寻道长度: 55.3	

图4-5 FCFS调度算法示例

磁道与磁头当前所在磁道的距离较近，这种新进程的I/O请求必被优先满足。对SSTF算法略加修改后所形成的SCAN算法，即可防止老进程出现饥饿现象。

SCAN算法不仅考虑到欲访问的磁道与当前磁道的距离，更优先考虑的是磁头的当前移动方向。例如，当磁头正在自里向外移动时，SCAN算法所选择的下一个访问对象应是其欲访问的磁道既在当前磁道之外，又是距离最近的。这样自里向外地访问，直到再无更外的磁道需要访问时，才将磁臂换向，自外向里移动。这时，同样也是每次选择这样的进程来调度，即其要访问的磁道，在当前磁道之内，从而避免了饥饿现象的出现。由于这种算法中磁头移动的规律颇似电梯的运行，故又称为电梯调度算法。图4-7显示按SCAN算法对9个进程进行调度及磁头移动的情况。

(从 100 磁道开始)

被访问的下一个磁道号	移动距离 (磁道数)
90	10
58	32
55	3
39	16
38	1
18	20
150	132
160	10
184	24
平均寻道长度: 27.5	

图4-6 SSTF调度算法示例

(从 100 磁道开始)

被访问的下一个磁道号	移动距离 (磁道数)
150	50
160	10
184	24
90	94
58	32
55	3
39	16
38	1
18	20
平均寻道长度: 27.8	

图4-7 SCAN调度算法示例

2) 循环扫描 (Circular SCAN, CSCAN) 算法。SCAN算法既能获得较好的寻道性能，又能防止进程饥饿，故被广泛用于大、中、小型机和网络中的磁盘调度。但也存在这样的问题：当磁头刚从里向外移动过某一磁道时，恰有一进程请求访问此磁道，这时该进程必须等待，待磁头从里向外，然后再从外向里扫描完所有要访问的磁道后，才处理该进程的请求，致使该进程的请求被严重地推迟。为了减少这种延迟，CSCAN算法规定磁头单向移动。例如，只自里向外移动，当磁头移到最外的被访问磁道时，磁头立即返回到最里的欲访磁道，即将最小磁道号紧接着最大磁道号构成循环，进行扫描。采用循环扫描方式后，上述请求进程的请求延迟，将从原来的 $2T$ 减为 $T + smnx$ 。其中， T 为由里向外或由外向里扫描完所有要访问的磁道所需的寻道时间，而 $smnx$ 是将磁头从最外而被访问的磁道直接移动最里边欲访问的磁道所需的寻道时间（或相反）。图4-8示出了CSCAN算法对9个进程调度的次序及每次磁头移动的距离。

(从 100 磁道开始)

被访问的下一个磁道号	移动距离 (磁道数)
150	50
160	10
184	24
18	166
38	20
39	1
55	16
58	3
90	32
平均寻道长度: 27.5	

图4-8 CSCAN调度算法示例

3. 廉价冗余磁盘阵列RAID

磁盘系统中比较引人注目的是廉价冗余磁盘阵列(Redundant arrays of inexpensive disk, RAID)的发展,这是将并行处理原理引入磁盘系统。它采用低成本的小温盘,使多台磁盘构成磁盘阵列,数据展开存储在多台磁盘上,提高数据传输的带宽,并利用冗余技术提高可靠性。磁盘阵列还具有容量大,数据传输率高,功耗低,体积小,成本低和便于维护等优点。

1987年美国加州大学伯克利分校的D.A.Patterson等人,首先提出了廉价冗余磁盘阵列的概念,并将RAID分为6级:

- RAID-0。该级仅提供了并行交叉存取。它虽然有效提高了磁盘I/O速度,但并无冗余校验功能,致使磁盘系统的可靠性不好。只要阵列中有一个磁盘损坏,便会造成不可弥补的数据丢失。
- RAID-1。它是镜像磁盘冗余阵列,将每一数据块重复存入镜像磁盘,以改善磁盘机的可靠性。镜像盘也称拷贝盘,它相当于一个不断进行备份操作的磁盘。这种磁盘的冗余度为100%,使有效容量下降了一半,成本较高。镜像盘是磁盘阵列的简单形式。
- RAID-2。它是采用海明码纠错冗余的磁盘阵列,将数据位交叉写入几个磁盘中,并利用几个磁盘驱动器进行按位的出错检查,它比镜像磁盘冗余阵列的冗余度小。这种阵列中的数据读写操作涉及阵列中的每一个磁盘,这影响小文件的传输率,因此它适合于大量顺序数据访问。
- RAID-3。它是采用奇偶校验冗余的磁盘阵列,也采用数据位交叉,阵列中只有一个校验盘。将数据按位交叉写到几个磁盘上,用一个校验盘检查数据错误。各磁盘同步运转,阵列中的驱动器数量可扩展。这种阵列冗余度较小,因为采用数据位交叉,所以也适合大量顺序数据访问。
- RAID-4。它是一种独立传送磁盘阵列,采用数据块交叉,用一个校验盘。将数据按块交叉存储在多个磁盘上。在数据不冲突时,多个磁盘可并行进行数据读操作。这种磁盘阵列适用于小块数据读写,它的小块数据传输速度比RAID-3快。
- RAID-5。它也是一种独立传送磁盘阵列,采用数据块交叉和分布的冗余校验,将数据和校验都分布在各个磁盘中,没有专门的奇偶校验驱动器。奇偶校验码被分布存放在阵列中各驱动器中,磁盘冗余度低,使并行读写操作成为可能。这种方法也适用于小块数据的读写。但对控制器的要求较高,是最难实现的一种磁盘阵列。

RAID自1988年面世后,很快流行起来,这主要是因为RAID具有以下明显的优点:

- 可靠性高。RAID的最大特点就是它的高可靠性。除了RAID-0级外,其余各级都采用了容错技术。与单台磁盘机相比,其可靠性往往高出一个数量级。
- 磁盘I/O速度高。由于磁盘阵列采取并行交叉存取,故可将磁盘I/O速度提高N-1倍,N为磁盘数目。
- 性能/价格比高。利用RAID技术实现大容量高速存储器时,其体积与相同容量和速度的大型磁盘系统相比,只是后者的三分之一,价格也是后者的三分之一,且可靠性更高。

4.1.4 高速缓存管理

目前,几乎所有可随机存取的文件,都是存放在磁盘上,磁盘I/O速度的高低,将直接影响

文件系统的性能。磁盘的I/O速度远低于对内存的访问速度，通常要低4~6个数量级。因此，磁盘的I/O已成为计算机系统的瓶颈。于是，人们便千方百计地去提高磁盘的速度，其中最主要的技术，便是利用高速缓存（cache）。

1. 磁盘高速缓存的形式

这里所说的磁盘高速缓存，并非通常意义下在内存和CPU之间所增设的一个小容量高速存储器，而是指利用内存中的存储空间，来暂存从磁盘中读出的一系列盘块中的信息。因此，这里的高速缓存是一组在逻辑上属于磁盘，而物理上是驻留在内存中的盘块。

高速缓存在内存中可分成两种形式：

- 在内存中开辟一个单独的存储空间来作为磁盘高速缓存，其大小是固定的，不会受应用程序多少的影响。
- 把所有未利用的内存空间变为一个缓冲池，供请求分页系统和磁盘I/O（作为磁盘高速缓存）共享。此时高速缓存的大小，显然不再是固定的。当磁盘I/O的频繁程度较高时，该缓冲池可能包含更多的内存空间；而在应用程序运行得较多时，该缓冲池可能只剩下较少的内存空间。

2. 数据交付

数据交付（data delivery）是指将磁盘高速缓存中的数据传送给请求者进程。当有一进程请求访问某个盘块中的数据时，由操作系统先去查看磁盘高速缓冲器，其中是否存在进程所需访问的盘块数据的拷贝。若有其拷贝，便直接从高速缓存中提取数据交付给请求者进程，这样，就避免了访盘操作，从而使本次访问速度提高4~6个数量级；否则，应先从磁盘中将所要访问的数据读入并交付给请求进程，同时也将数据送高速缓存，当以后又需要访问该盘块的数据时，便可直接从高速缓存中提取。

系统可以采取下述两种方式，将数据交付给请求者进程。

- 数据交付：这是直接将高速缓存中的数据，传送到请求者进程的内存工作区中。
- 指针交付：只将指向高速缓存中某区域的指针，交付给请求者进程。

后一种方式由于所传送的数据少，因而节省了数据从存储器到存储器的时间。

3. 置换算法

如同请求调页（段）一样，在将磁盘中的盘块数据读入高速缓存时，同样会出现因高速缓存中已装满盘块数据，固需要将高速缓存中的数据先换出的问题。相应地，也存在着采用哪种置换算法的问题。较常用的置换算法仍然是最近最久未使用（LRU）算法、最近未使用（NRU）算法及最不常用（LFU）算法等。

由于请求调页中的关联存储器与高速缓存（磁盘I/O）的工作情况不同，因而使得在转换算法中所应考虑的总是也有所差异。因此，现在不少系统在设计高速缓存的置换算法时，除了考虑到最近最久未使用这一原则外，还考虑了以下几点：

- 访问频率。通常，每执行一条指令时，便可能访问一次关联存储器，亦即关联存储器的访问频率，基本上与指令的执行频率相当；而对高速缓存的访问频率，则与磁盘I/O的频率相当，因此，对关联存储器的访问频率远远高于对高速缓存的访问频率。
- 可预见性。在高速缓存中的各盘块数据，有哪些数据可能在较长时间内不会再被访问，又

有哪些数据可能很快就再被访问，会有相当一部分是可预知的。例如，对二级地址块及目录块等，在它被访问后，可能会很久都不再被访问。又如，正在写入数据的未满足块，可能会很快又被访问。

- 数据的一致性。由于高速缓存是处在内存中，而内存一般说来又是一种易失性的存储器，一旦系统发生故障，存放在高速缓存中的数据将会丢失，而其中有些盘块（如索引结点盘块）中的数据已被修改，但尚未拷回磁盘。因此，当系统发生故障后，可能会造成数据的不一致性。

基于上述考虑，在有的系统中便将高速缓存中的所有盘块数据拉成一条LRU链。对于那些会严重影响数据一致性的盘块数据和很久都可能不再使用的盘块数据，都放在LRU链的头部，使它们能被优先写回磁盘，以减少发生数据不一致性的概率，或者可以尽早地腾出高速缓存的空间。对于那些可能在不久之后便要使用的盘块数据，应挂在LRU链的尾部，以便在不久以后需要时，只要该数据块尚未从链中移至链首而被写回磁盘，便可直接到高速缓存中（即LRU链中）去找到它们。

4. 周期性写回磁盘

还有一种情况值得注意：那就是根据LRU算法，那些经常被访问的盘块数据，可能会一直保留在高速缓存中而长期地不会被写回磁盘中。（注意，LRU链意味着链中的任一元素在被访问之后，总是又被挂到链尾而不被写回磁盘；只是一直未被访问的元素，才有可能移动到链首，而被写回磁盘。）例如，一位学者一上班便开始撰写论文，边写边修改，他正在写作的论文就一直保存在高速缓存的LRU链中。如果在快下班时，系统突然发生故障，这样，存放在高速缓存中的已写论文，将随之消失，致使他枉费了一天的劳动。

为了解决这一问题，在UNIX系统中专门增设了一个修改（update）程序，使之在后台运行。该程序周期性地调用一个系统调用SYNC。该调用的主要功能是强制性地将所有在高速缓存中已修改的盘块数据写回磁盘，一般是把两次调用SYNC的时间间隔定为30秒钟。这样，因系统故障所造成的工作损失不会超过30 s的劳动量。而在MS-DOS中所采用的方法是：只要高速缓存中的某盘块数据被修改，便立即将它写回磁盘，并将这种高速缓存称为“通写高速缓存”（write-through cache）。MD-DOS所采用的写回方式，几乎不会造成数据的丢失，但需频繁地启动磁盘。后面会详细介绍Windows 2000/XP的解决方式。

5. 提高磁盘I/O速度的其他方法

在系统中设置了磁盘高速缓存后，能显著地减少等待磁盘I/O的时间。下面介绍几种能有效地提高磁盘I/O速度的方法，这些方法已被许多系统，如Windows 2000/XP，所采用。

1) 预读 用户（进程）对文件进行访问时，经常采用顺序访问方式，即顺序地访问文件的各盘块的数据。在这种情况下，在读当前块时可以预知下一次要读的盘块，因此，可以采取预读（read-ahead）方式。即在读当前块的同时，还要求提前将下一个盘块（提前读的块）中的数据也读入缓冲区。这样，当下一次要读该盘块中的数据时，由于该数据已被提前读入缓冲区，因而此时便可直接从缓冲区中取得下一盘块的数据，而不须再去启动磁盘I/O，从而大大减少了读数据的时间。这也就等效于提高了磁盘I/O的速度。“预读”功能已被广泛应用，如在Windows 2000/XP、UNIX和OS/2等操作系统中都已采用。

2) 延迟写。延迟写是指在缓冲区A中的数据本应立即写回磁盘,但考虑到该缓冲区中的数据,不久之后可能还会再被本进程或其他进程访问(共享数据),因而并不立即将该缓冲区A中的数据写入磁盘,而是将它挂在空闲缓冲区队列的末尾。随着空闲缓冲区的使用,缓冲区A也慢慢往前移动,直至移动到空闲缓冲区队列之首。当再有进程申请到该缓冲区时,才将该缓冲区中的数据写入磁盘,而把该缓冲区作为空闲缓冲区分配出去。当该缓冲区A仍在队列中时,任何访问该数据的进程,便可直接读出其中的数据,而不必去访问磁盘。这样,又可进一步减小等效的磁盘I/O时间。同样,“延迟写”功能已在Windows 2000/XP、UNIX、OS/2等操作系统中被广泛地采用。

3) 虚拟盘。所谓虚拟盘(virtual disk),是指利用内存空间去仿真磁盘,又称为RAM盘。该盘的设备驱动程序可以接受所有标准的磁盘操作,但这些操作的执行,不是在磁盘上而是在内存中。这些对用户都是透明的。换言之,用户们并不会发现这与真正的磁盘操作有什么不同,而仅仅是略微快些而已。虚拟盘的主要问题是:它是易失性存储器,故一旦系统或电源发生故障,或系统再启动时,原来保存在虚拟盘中的数据将会丢失。因此,虚拟盘通常用于存放临时文件,如编译程序所产生的目标程序等。虚拟盘与磁盘高速缓存的主要区别在于:虚拟盘中的内容完全由用户控制,而磁盘高速缓冲区中的内容是由OS控制的。例如,RAM盘在开始时是空的,只有当用户(程序)在RAM盘中创建了文件后,RAM盘中才有内容。

4.2 Windows 2000/XP内存管理

内存管理器是Windows 2000/XP执行体的一部分,位于Ntoskrnl.exe文件中。在硬件抽象层(HAL)中没有内存管理器的任何部分。它由以下几个部分组成:

- 一组执行体系统服务程序,用于虚拟内存的分配、回收和管理。大多数这些服务都是以Win32 API或核心态的设备驱动程序接口形式出现。
- 一个转换无效和访问错误陷阱处理程序,用于解决硬件检测到的内存管理异常,并代表进程将虚拟页面装入内存。
- 运行在六个不同的核心态系统线程上下文中的几个关键组件:
 - 工作集管理器(working set manager)(优先级为16) 平衡集管理器(内核创建的系统线程)每秒钟调用它一次。当空闲内存低于某一界限时,便启动所有的内存管理策略,如工作集的修整、老化和已修改页面的写入等。
 - 进程/堆栈交换程序(process/stack swapper)(优先级为23) 完成进程和内核线程堆栈的换入和换出操作。当需要进行换入和换出操作时,平衡集管理器和内核中的线程调度代码将唤醒该线程。
 - 已修改页面写入器(modified page writer)(优先级为17) 将修改链表上的“脏”页写回到适当的页文件。需要减小修改链表的大小时,此线程将被唤醒。
 - 映射页面写入器(mapped page writer)(优先级为17) 将映射文件中脏页写回磁盘。需要减小修改链表的大小,或映射文件中某些页面在修改链表中超过了5分钟时,它将被唤醒。
 - 废弃段线程(dereference segment thread)(优先级为18) 负责系统高速缓存和页面文件的扩大和缩小。(例如,如果没有虚拟地址空间满足分页缓冲池的增加,该线程将减小系

统高速缓存的大小。)

- 零页线程(zero page thread) (优先级为0) 将空闲链表中的页面清零, 以便有足够的零页面满足将来的零页需求。

正如其他所有的Windows 2000/XP 执行程序组件一样, 内存管理器是完全可重入的, 它支持多进程并发执行。为了实现可重入, 内存管理器使用了几个不同的内部同步机制来控制它自身数据结构的访问, 如旋转锁和执行程序资源。

下面将分别介绍Windows 2000/XP内存管理系统, 包括进程虚存空间的布局、基于Intel x86 体系结构的地址变换过程、分配和回收虚拟内存的系统服务、工作集机制和物理内存的管理, 着重描述内存管理机构的组件、关键的数据结构以及相应的算法。最后, 还介绍了内存保护、写时复制以及地址窗口扩展等技术。

4.2.1 地址空间的布局

默认情况下, 32位Windows 2000/XP上每个用户进程可以占有2GB的私有地址空间(address space); 操作系统占有剩下的2GB地址空间。Windows 2000/XP高级服务器和Windows 2000/XP数据中心服务器支持一个引导选项, 允许用户拥有3GB的地址空间。这两个地址空间的布局如图4-9所示。3GB地址空间选项(在Boot.ini 中通过/3GB标识激活) 提供进程一个3GB的地址空间(剩下1GB作为系统空间)。这个特性是为满足一些应用程序的需求而采用的临时解决办法。例如, 数据库服务器需要在内存中保存比2GB地址空间更多的数据。

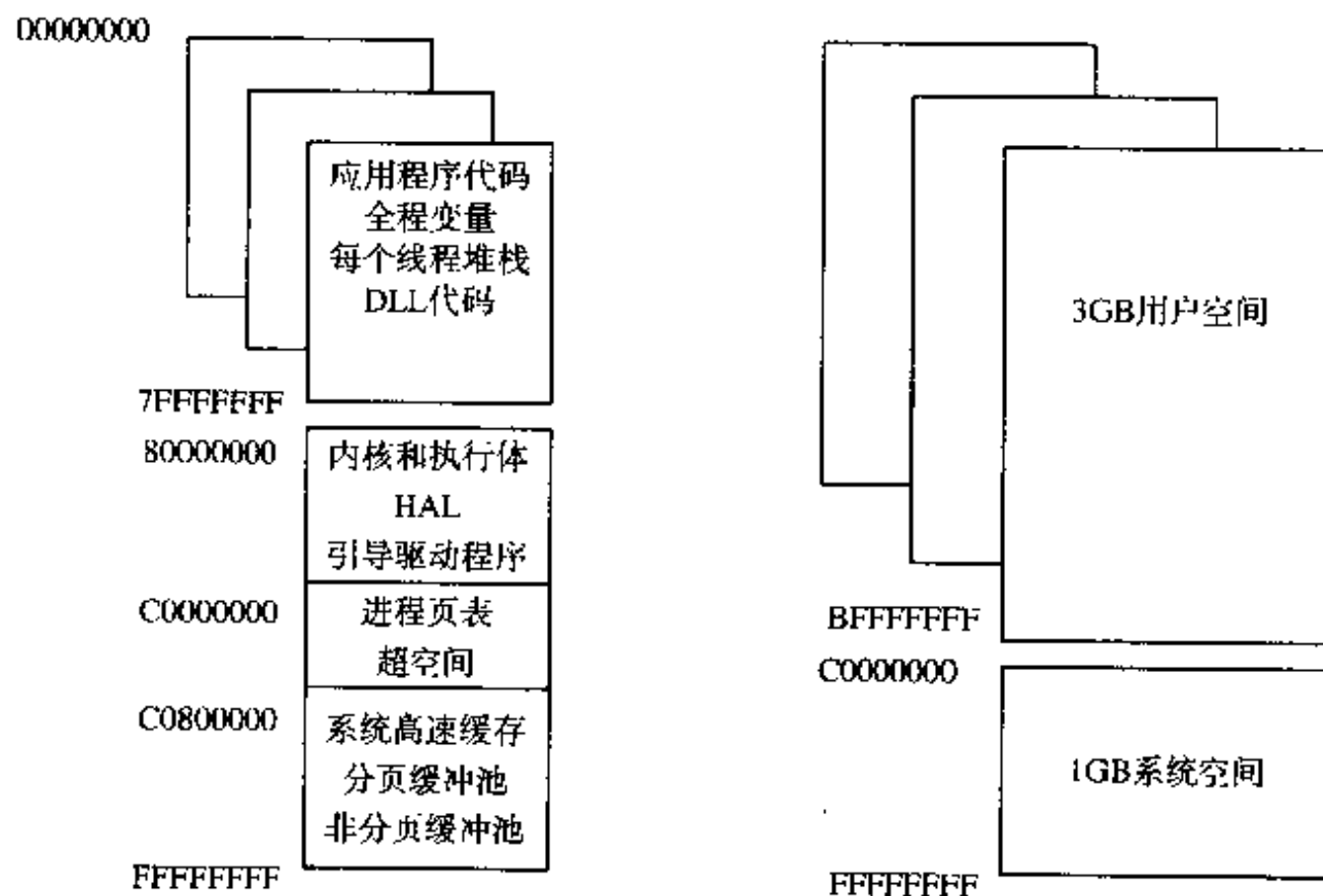


图4-9 x86系统虚拟地址空间布局

对于需要访问整个3GB地址空间的进程而言, 进程映像文件必须在映像头设置IMAGE-FILE-LARGE-ADDRESS-AWARE标识, 否则, Windows 2000/XP将保留第3个GB的地址空间, 而应用程

序不会看到大于0x7FFFFFFF的虚拟地址空间。可以通过指定链接标识/LARGEADDRESSAWARE来设置该标识。在2GB用户地址空间的系统上运行应用程序时，这个标识没有影响。

1. 用户地址空间分布

表4-1详细描述了2GB Windows 2000/XP用户进程地址空间的布局。表4-2显示的系统变量定义了用户地址空间的范围。表4-3列出了性能计数器提供的有关全部系统虚拟内存使用的信息。表4-4列出了进程性能计数器得到的单个进程地址空间使用情况。

表4-1 2GB Windows 2000/XP用户进程地址空间的布局

范 围	大 小	功 能
0x0~0xffff	64KB	拒绝访问区域，帮助程序员避免不正确的指针引用，试图访问该区域的地址将导致访问违规（该地址区域习惯上用来帮助查找错误）
0x10000~0x7FFEFFFF	2GB减去至少192KB	进程私有地址空间
0x7FFDE000~0x7FFDEFFF	4KB	第一个线程的线程环境块（TEB）。在此页的前面创建附加的TEB（从地址0x7FFDD 000开始向后延伸）
0x7FFDF000~0x7FFDFFFF	4KB	进程环境块（PEB）
0x7FFE0000~0x7FFE0FFF	4KB	共享的用户数据页面。该只读数据页面映射到系统空间中包括系统时间、时钟计数和版本号信息的一页。该页面的存在使得这些数据可以直接从用户态读取面不用要求内核模式转换
0x7FFE1000~0x7FFEFFFF	60KB	拒绝访问区域
0x7FFF0000~0x7FFFFFFF	64KB	拒绝访问区域，阻止线程跨过用户/系统边界传送缓冲区。MmUserProbeAddress中包含该区域的起始地址

表4-2 Windows 2000/XP 用户地址空间系统变量

系 统 变 量	描 述	x86 2GB用户空间	x86 3GB用户空间
MmHighestUserAddress	最高用户地址(因为TEB和PEB的存在最高可用地址实际上要小一些)	0x7FFEFFFF	0xBFFEFFFF
MmUserProbeAddress	最高用户地址+1（用于探求用户缓冲区的可访问性）	0x7FFF0000	0xBFFF0000

表4-3 Windows 2000/XP 虚拟内存使用性能计数器

性能计数器	系 统 变 量	描 述
Memory: Committed Bytes	MmTotalCommittedPages	提交的私有地址空间数量（一些在物理内存，一些在页文件里）
Memory: Committed Limit	MmTotalCommitLimit	在不增加页文件（页文件是可扩充的）大小的情况下，可以提交的内存字节数
Memory: %Committed Bytes in Use	MmTotalCommittedPaged/ MmTotalCommitLimit	提交字节和提交限制之比

表4-4 Windows 2000/XP单进程性能计数器的地址空间使用

性能计数器	描 述
Process: Virtual Bytes	进程地址空间全部大小 (包括共享的和私有的页面)
Process: Private Bytes	私有页面 (非共享) 占用的地址空间大小 (同Process: Page File Bytes)
Process: Page File Bytes	私有页面 (非共享) 占用的地址空间大小 (同Process: Private Bytes)
Process: Peak Page File Bytes	Process: Page File Bytes的峰值

2. 系统地址空间分布

本段详细描述系统空间的布局和内容。图4-10显示了在x86系统上2GB系统空间的全部结构。

80000000	系统代码(Ntoskrnl, HAL) 和系统中一些初始的未分页缓冲池
A0000000	系统映射视图 (例如, Win32k.sys) 或者会话空间
A4000000	附加的系统PTS (高速缓存可以扩展到这里)
C0000000	进程的页表和页目录
C0400000	超空间和进程工作集列表
C0800000	没有使用, 不可访问
C0C00000	系统工作集列表
C1000000	系统高速缓存
E1000000	分页缓冲池
EB000000 (min)	系统PTE
	未分页缓冲池扩充
FFBE0000	故障转储信息
FFC00000	HAL使用

图4-10 x86系统空间布局 (不是成比例的)

X86体系结构下系统空间由以下组成部分:

- 系统代码 包括操作系统映像、HAL和用于引导系统的设备驱动程序。
- 系统映射视图 用来映射Win32子系统可加载的核心态部分Win32k.sys, 以及它使用的核心态图形驱动程序。
- 会话空间 (session space) 用来映射一个用户的会话信息。(在装载终端服务时, Windows 2000/XP可以支持多用户会话。) 会话工作集链表描述了常驻或正在使用中的部分会话空间。
- 进程页表(page table)和页目录(page directory) 描述虚拟地址映射的结构。
- 超空间 (hyperspace) 一个特殊的区域用来映射进程工作集链表, 并为下列操作临时映射物理页面: 将空闲链表上的页置零 (当零链表为空而且需要一个零页时), 使其他页表的页表项无效 (例如当从备用链表中删除一个页面时), 以及在进程创建时建立一个新的进程地址空间。

- **系统工作集链表** 描述系统工作集的工作集链表数据结构。(在微软内部,系统工作集叫做系统高速缓存工作集。但这是一个容易引起误解的名词,因为它不仅包括系统高速缓存,还包括分页缓冲池,可分页的系统代码和数据,可分页的驱动程序代码和数据。因此,本章将它称为系统工作集。)
- **系统高速缓存(system cache)** 用来映射在系统高速缓存中打开的文件的虚拟空间。
- **分页缓冲池(paged pool)** 可分页系统内存堆。
- **系统页表项(Page table entries, PTE)** 系统PTE缓冲池,用来映射系统页面,例如I/O空间、内核栈和内存描述符表。在“性能监视器”中检查Memory:Free System PTE计数器的值,可以查看有多少系统PTE是可用的。
- **非分页缓冲池(nonpaged pool)** 不可分页的系统内存堆,通常存在两个地方——一部分在系统空间高端,一部分在低端。
- **系统性故障转储信息** 被保留来记录关于系统性故障的状态信息。
- **HAL使用区域** 为HAL特定的结构而保留的系统内存。

下面两个表列出了系统空间的详细结构。表4-5列出了内核变量,包括各种系统空间区域开始和结束的地址。其中一些区域是固定的;一些系统空间是当系统启动时,根据内存大小和运行的是专业Windows 2000/XP或Windows 2000/XP服务器的基础上计算出来的。表4-6列出了在x86体系结构上系统空间的结构。

表4-5 描述系统空间区域的系统变量

系统变量	描述	x86 2GB系统空间 (非PAE)	x86 1GB系统空间 (非PAE)
MmSystemRangeStart	系统空间起始地址	0x80000000	0xC0000000
MmSystemCacheWorkingSetList	系统工作集链表	0xC0C00000	0xC0C00000
MmSystemCacheStart	系统缓存起始地址	0xC1000000	0xC1000000
MmSystemCacheEnd	系统缓存结束地址	计算出	计算出
MiSystemCacheStartExtra	系统缓存起始地址或系统PTE扩展	0xA4000000	0
MiSystemCacheEndExtra	系统缓存结束地址或PTE扩展	0xC0000000	0
MmPagedPoolStart	分页缓冲池起始地址	0xE1000000	0xE1000000
MmPagedPoolEnd	分页缓冲池结束地址	计算出(最大482MB)	计算出(最大482MB)
MmNonPagedSystemStart	系统PTE起始地址	计算出(最低0xEB000000)	计算出
MmNonPagedPoolStart	非分页缓冲池起始地址	计算出	计算出
MmNonPagedPoolExpansionStart	扩展非分页缓冲池起始地址	计算出	计算出
MmNonPagedPoolEnd	非分页缓冲池结束地址	0xFFBE0000	0xFFBE0000

表4-6 x86系统空间（非PAE）

地址范围	大 小	功 能
0x8000000~0x9FFFFFFF	512MB	引导系统（Ntoskrnl.exe和Hal.dll）和非分页缓冲池初始部分的系统代码。在2GB系统空间和128MB以上RAM的x86系统上，最初的512MB是使用x86大页PDE映射的
0xA0000000~0xA2FFFFFF	48MB	如果没安装终端服务，则是系统映射视图，否则是会话空间（见表4-7）
0xA3000000~0xA3FFFFFF	16MB	终端服务的系统映射视图
0xA4000000~0xBFFFFFFF	448MB	附加系统PTE(用于内核栈，映射I/O空间等)或附加系统高速缓存（用于大高速缓存的系统）
0xC0000000~0xC03FFFFFF	4MB	进程页表(页目录在0xC0300 0000,大小为4KB)。这是被映射到系统空间的每个进程的数据
0xC0400000~0xC07FFFFFF	4MB	工作集链表和超空间。这是被映射到系统空间的每个进程的数据
0xC0800000~0xC0BFFFFFF	4MB	未使用
0xC0C00000~0xC0FFFFFF	4MB	系统工作集链表
0xC1000000~0xE0FFFFFF	512MB（最大）	系统高速缓存（在引导时计算大小）
0xE1000000~0xEAFFFFFFFF*	160MB（最大）	分页缓冲池（在引导时计算大小）
0xEB000000~0xFFBDBFFF	331.875MB (339 840KB)	系统PTE和非分页缓冲池（在引导时计算大小）。如果注册表中PagedPoolSize值被设置成-1，系统PTE从0xEB000000地址范围移到0xA4000000地址范围，分页缓冲池可以使用这块地址空间
0xFFBE0000~0xFFFFFFFF	4.125MB(4224KB)	故障转储结构和HAL数据结构

* 因为分页缓冲池被非分页缓冲池和系统PTE的区域起始地址限制，只有那些地址不被使用时，它可以处于地址0xEB000000之上。

表4-7 会话空间布局

地址范围	大 小	功 能
0xA0000000~0xA07FFFFFF	8MB	Win32k.sys和Windows NT4打印驱动程序
0xA0800000~0xA0BFFFFFF	4MB	MM-SESSION-SPACE结构和会话工作集链表
0xA0C00000~0xA1FFFFFF	20MB	此会话的映射视图
0xA2000000~0xA2FFFFFF	16MB	此会话的分页缓冲区

3. 会话空间

会话是由进程和其他系统对象（例如窗口工作站、桌面和窗口）组成，这些系统对象代表了一个登录到会话的单用户工作站。Win32子系统（Win32k.sys）核心态部分使用一个特定分页缓冲池区域，为每个会话分配私有GUI数据结构。另外，每个会话有自己的Win32子系统进程（Csrss.exe）和登录进程（Winlogon.exe）的拷贝。会话管理器进程（Smss.exe）负责创建新的会话，包括加载Win32k.sys的会话私有拷贝，创建会话私有对象管理器名字空间，以及生成特定的

Csrss和Winlogon进程实例。

为了实现会话，所有与会话相关的数据结构都被映射到一块称为会话空间的系统空间中，地址从0xA0000000开始扩展到0xA2FFFFFF。当进程创建时，这一地址区域被映射到属于该进程会话的页面。表4-7列出了安装终端服务的系统上会话空间的布局。

4.2.2 地址转换机制

通过上一节，已经了解Windows 2000/XP是如何组织32位虚拟地址空间的。下面将介绍它是如何将虚拟地址空间映射到真实的物理页面上。我们还将描述，当地址变换找不到物理存储器的地址时，系统会如何处理，并解释Windows 2000/XP是如何通过工作集和页框数据库来管理物理内存的。

用户应用程序以32位虚拟地址方式编址。CPU利用内存管理器创建和维护的数据结构将虚拟地址变换为物理地址。图4-11是三个连续的虚拟页面映射到三个不连续的物理页面的示意图。

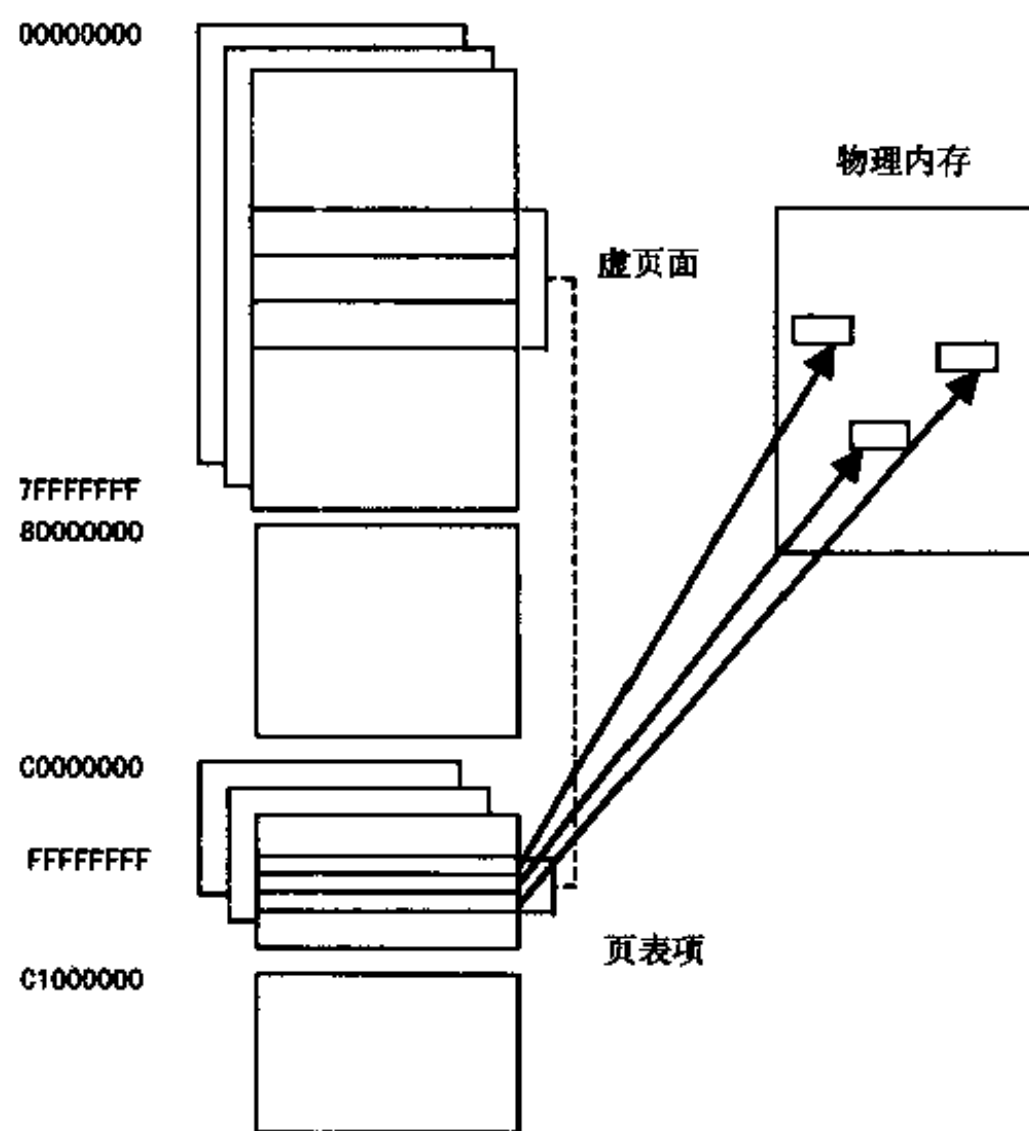


图4-11 虚拟地址映射到物理内存

图4-11中，连接虚拟页面和页表项的虚线表明虚拟页面和物理内存之间的间接关系。虚拟地址不直接映射到物理地址，而是每个虚拟地址都与一个称作“页表项”（PTE）的结构有关，而虚拟地址映射的物理地址就包含在这个结构中。下面，我们将详细解释Windows 2000/XP是如何实现上述地址映射的问题。

1. 虚拟地址变换

Windows 2000/XP在x86体系结构上利用二级页表结构来实现虚拟地址向物理地址的变换。(运行物理地址扩展(PAE)内核的系统是利用三级页表——下面的讨论假定系统为非PAE系统。)一个32位虚拟地址被解释为三个独立的分量——页目录索引、页表索引和字节索引——它们用于找出描述页面映射结构的索引。如图4-12所示,页面大小及页表项的宽度决定了页目录和页表索引的宽度。比如,在x86系统中,因为一页包含4096字节,于是字节索引被确定为12位宽($2^{12} = 4096$)。

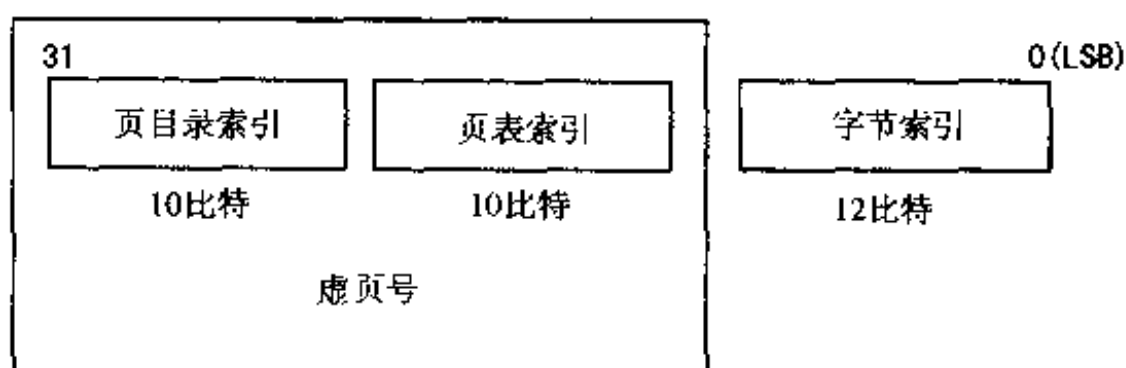


图4-12 x86系统中一个32位虚拟地址的构成

“页目录索引”用于指出虚拟地址的页目录在页表中的位置。“页表索引”则用来确定页表项在页表中的具体位置。如前所述,页表项包含了虚拟地址被映射到的物理地址。“字节索引”使我们能在物理页中寻找某个具体的地址。图4-13表示了这三个值之间的联系和它们在虚拟地址到物理地址的映射过程中所起的作用。

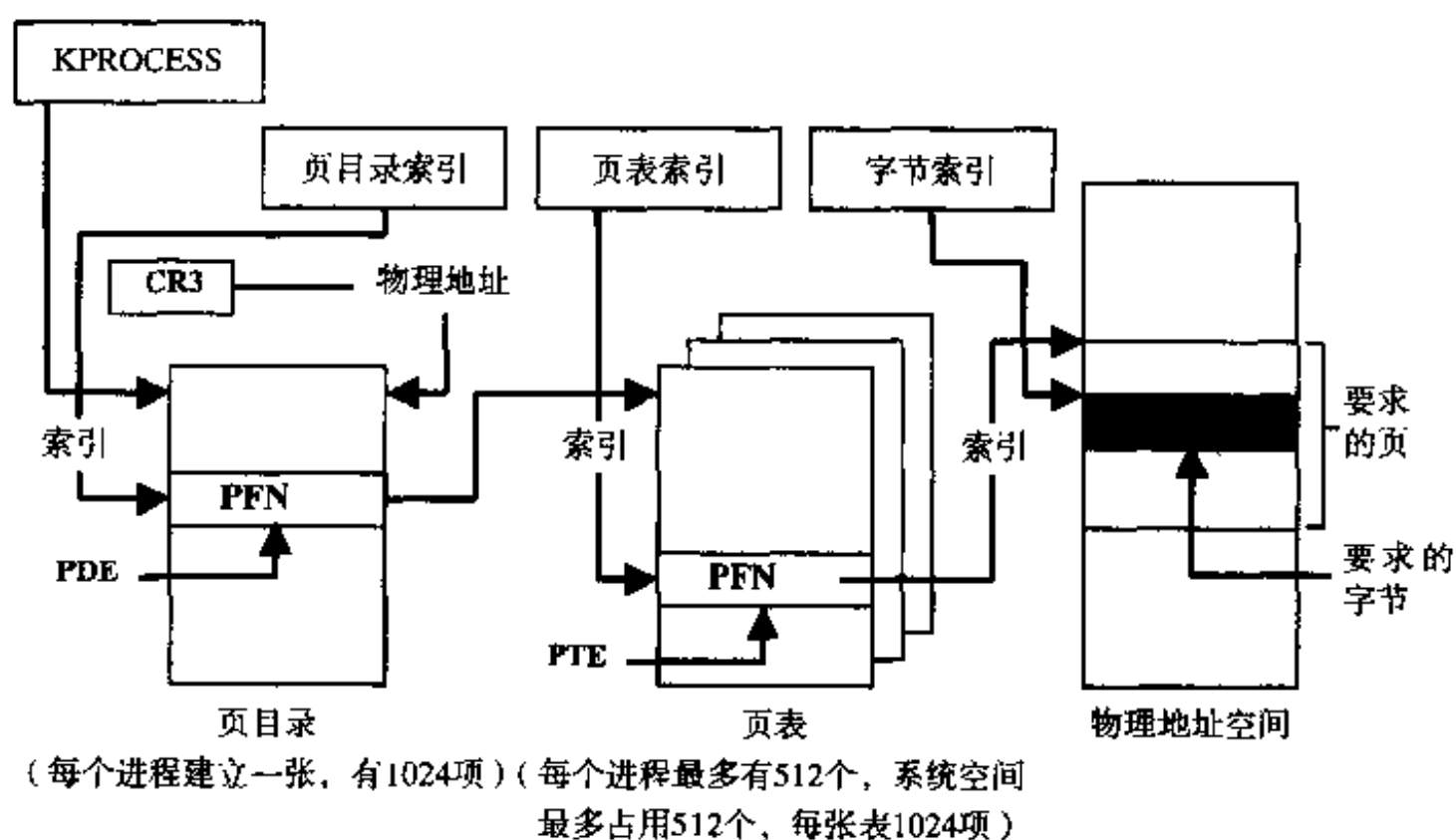


图4-13 虚拟地址的变换 (x86系统)

下面是一个虚拟地址变换的基本步骤:

1) 内存管理的硬件设备定位当前进程的页目录。每次进程的切换时,一般是通过操作系统设

置一个专用的CPU寄存器来通知硬件设备新进程页目录所在的地址。

2) 页目录索引用于在页目录中指出页目录项 (page directory entry, PDE) 的位置。页目录项包含的页框号 (page frame number, PFN) 描述了映射虚拟地址所需页表的位置。

3) 页表索引用于在页表中指明页表项的位置。页表项描述了虚拟页面在物理内存的位置。

4) 页表项用于确定页框的位置。如果所需的页是有效的, 页表项会包含物理内存中一个页的页框号。相应的虚拟页面就包含这个物理页框中。如果页表项表明所需的页是无效的, 内存管理器的故障处理程序会定位该页, 并努力试图使之有效。(详情参阅“缺页处理”部分。) 如果不能使失效的页面有效(比如是因为一个保护错误), 故障处理程序将产生一个访问违规或错误检查。

5) 当页表项指向了有效的页时, 字节索引用于找到物理页内所需数据的地址。

现在我们对页目录、页表和页表项已经有了一个总体印象。下面将了解它们的详细结构。

2. 页目录

每个进程都拥有一个单独的页目录, 这是由内存管理器创建的特殊页, 用于映射进程所有页表的位置。进程页目录的物理地址被保存在核心进程 (KPROCESS) 块中。实际上在x86系统中它还同时被映射到地址0xC0300000处(在运行PAE内核的系统中, 则映射到0xC06000000处)。

CPU之所以知道页目录页面的所在位置, 是因为CPU内部有一个专用寄存器 (x86系统中的CR3), 而操作系统将页目录的物理地址放在这个寄存器中。任何时刻一次进程切换均会产生一个不同于当前线程的其他线程。这个专用寄存器的内容会被新进程的核心进程块的相应信息刷新。因为同一进程的不同线程共享同一个进程地址空间, 所以同一进程的不同线程之间切换不会导致页目录物理地址的更新。

页目录是由页目录项 (PDE) 组成的, 每个页目录项4字节长 (在运行PAE内核的系统中则为8字节长), 描述了进程所有页表的状态和位置。(页表是根据需求创建的, 所以大多数进程的页目录仅指向页表的一小部分。) 页目录项的格式这里就不重复了, 因为它与硬件页表项大致相同。

在x86系统中, 需要1024张页表 (PAE系统是2048张) 来描述总共4GB的虚拟地址空间。进程的页目录将这些页表映射到1024个页目录项上。因此, 页目录索引需要有10位宽 ($2^{10} = 1024$)。

3. 进程页表与系统页表

在利用页内字节偏移引用一个字节之前, CPU首先需要能找到包含所需字节的页面。因此, 操作系统在内存中构造了另一个包含映射信息的页面。这个包含映射信息的页面就叫做“页表”。因为Windows 2000/XP为每个进程均提供一个私有的地址空间, 而且每个进程间的地址映射情况各异, 所以每个进程都有自己独有的页表集来映射私有地址空间。

描述系统空间的页表被所有的进程共享。当进程刚创建时, 系统空间的页目录项被初始化为指向现存的系统页表。但是如图4-14所示, 各个进程的系统空间不完全相同。例如, 如果分页缓冲池扩展请求分配一个新的系统页表, 内存管理器不会去更新所有的进程页目录, 使它们均指向新的系统页表, 而是只有当进程访问新的虚拟地址时才对进程页目录进行更新。

访问已驻留内存的分页缓冲池时, 进程也可能产生一个页面失效, 这是因为它的进程页目录

仍然没有指向描述缓冲池新区域的系统页表。当访问非分页缓冲池时页面失效不会发生，甚至尽管它也能被扩充，这是因为Windows 2000/XP在系统初始化时为了能描述它的最大容量已经建立了足够多的系统页表。

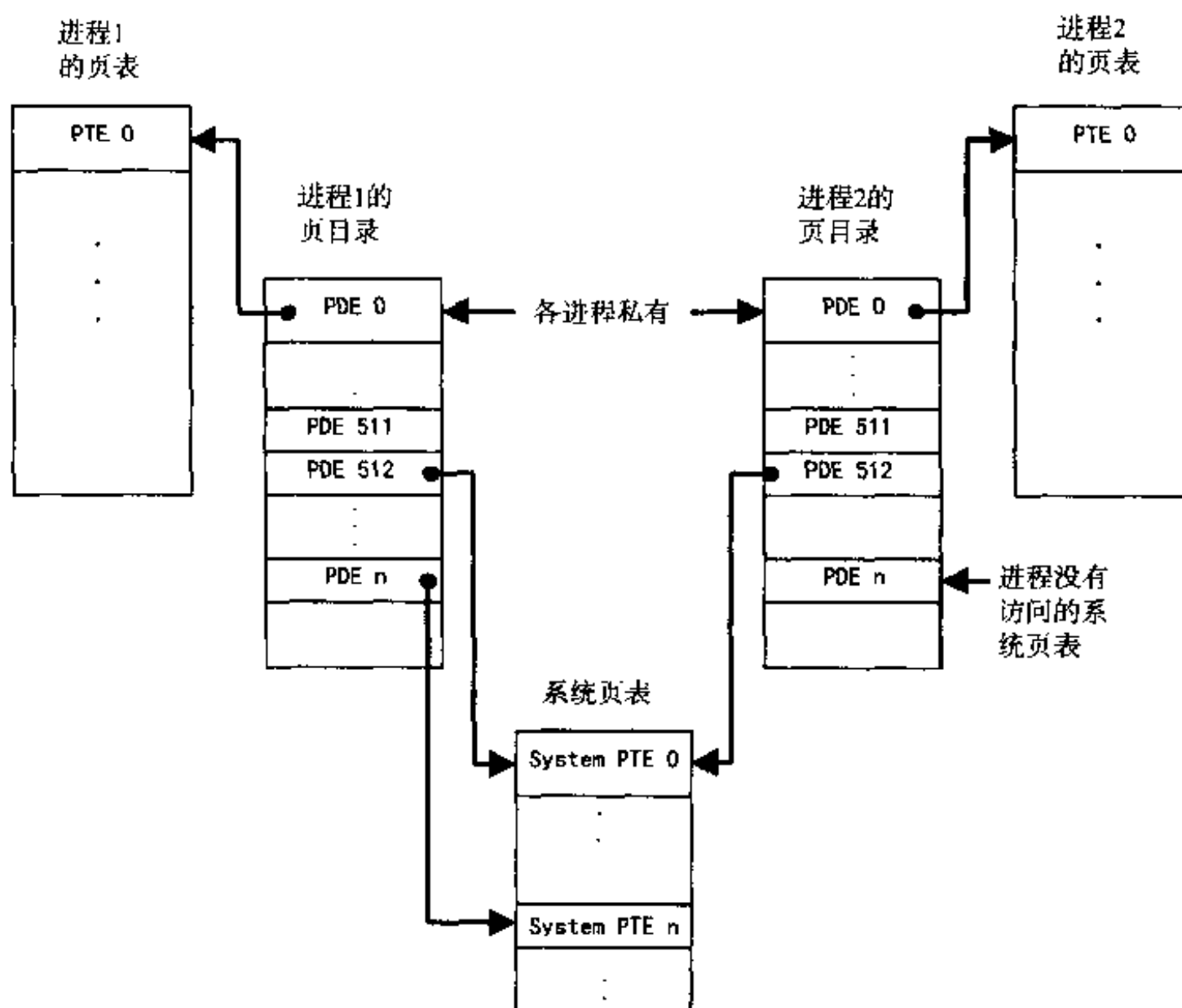


图4-14 系统页表与进程私有页表

系统页表项不是一种无限的资源——Windows 2000/XP基于内存容量来计算提供多少系统页表项用来分配。通过检查性能监视器中空闲系统页表项计数器的值，可以看到有多少系统页项是可用的，也可以将HKLM\SYSTEM\CurrentControlSet\Control\SessionManager\MemoryManagement\SystemPages设置为需要的页表项数量来覆盖启动时的计算结果。

4. 页表项

如前所述，页表是由页表项（PTE）数组构成的。可以利用kernel debugger中的“!pte”命令来分析页表项。有效的页表项（这是将要在这里讨论的，无效的页表项将在稍后的章节里介绍）有两个主要的域：包含数据的物理页面的页框号，或是内存中某页面的物理地址的页框号；另外是一些描述页的状态和保护限制的标志位，如图4-15所示。

图4-15中标注“保留”的位仅用于页表项为无效的时候（这些位由软件解释）。表4-8简要说明了一个有效页表项中由硬件规定的各位的具体含义。

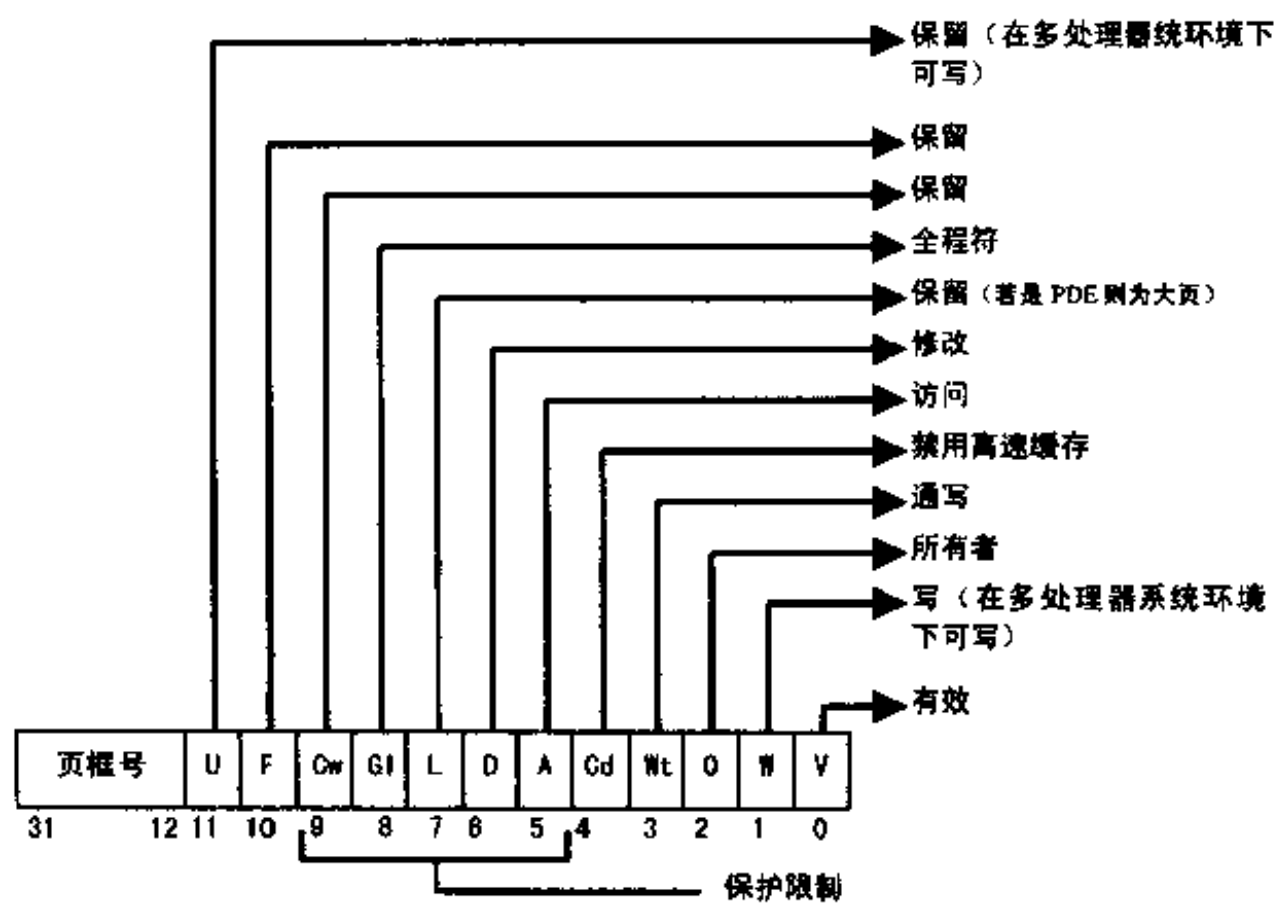


图4-15 有效的x86硬件页表项

表4-8 页表项状态位和保护限制位

标志位	含 义
访问	此页已被读过
禁用高速缓存	禁止访问此页的高速缓存
修改	此页已被写过
全程符	变换对全部进程有效（例如，变换缓冲区的刷新不会影响到这个页表项）
大页	在有128MB内存以上的系统中表示页目录项映射4 MB的页面（通常用于映射 Ntoskrnl和HAL、初始的非分页缓冲池等）
所有者	表明此页是否可以在用户态下访问，或仅可以在核心态下访问
有效	表示变换是否映射到物理内存中实际的页面
直接写	写入此页时禁用高速缓存，这样数据的修改能立刻刷新外存
写	在单处理器系统上，表示此页是可读写的或只读的；在多处理器系统上，表示此页是否可写（这时写位存储在页表项保留位上）

在x86系统中，硬件页表项包括一个修改位和一个访问位。如果物理页在页表项中表现为既没被读过也没被写过，那么访问位为零；当某页首次被读写时，处理器会将此位置“1”。而仅当某页首次被写时，修改位才会被处理器置“1”。除了这两个标志位，x86体系结构还有一个写位用来提供页的写保护——当此位为零时，对应的页为只读的；当此位为“1”时，对应的页是可读写的。如果一个页的写位为零时，一个线程试图对其进行写操作，将会引发内存管理的异常，内存管理器的访问故障处理程序（将在下节详述）必须确定这个线程能否对此页执行写操作（比如，如果此页确实标记为“写时复制”），或者是否应当产生一次访问违规。

在多处理器的x86系统中，硬件页表项还有一个附加的由软件实现的写位，主要是为了在不

同的处理器对页表项的快表（TLB）刷新时消除延迟。这位表示某页已经被一个运行在多个处理器上的线程写入。

在x86硬件平台上，页表项总是4个字节（可以运行PAE的系统为8字节），这样每张页表都包含1024个（PAE系统512个）页表项（每页4096字节，每个页表项4字节），而且因此可以映射1024个（PAE系统512个）4MB大小（PAE系统2MB）的页。

在x86系统中，页表索引为10位宽（PAE为9位），最多允许索引1024个页表项（PAE为512个）。Windows 2000/XP提供4GB的虚拟地址空间，所以需要多张页表来映射全部地址空间。x86系统中的每张页表可以映射4MB大小（PAE为2MB）的数据页，因此需要1024张页表（4GB/4MB）来映射4GB的地址空间。如果是PAE系统，则需要2048张页表（4GB/2MB）。

5. 字节索引

一旦内存管理器找到需要的物理页，它还必须在页内找到所需的数据。这就是引入字节索引域的用途。字节索引域告诉CPU数据在页内的偏移。在x86系统中，字节索引为12位宽，最多可以索引4096个数据字节（正是一页的大小）。

6. 快表TLB

每次地址变换都需要经过两次查询：一次是在页目录中找到正确的页表，另一次是在页表中找到正确的项。如果每次对虚拟地址的访问都做两次额外的内存查询，这将使系统性能严重下降。因此大多数CPU在地址变换时运用了高速缓存技术。x86处理器提供了关联存储器数组形式的高速缓存，称为快表（Translation Lookaside Buffer, TLB）。所谓关联存储器，例如TLB，是一个向量，它的存储单元能被同时读取，并与目标值比较。在TLB中，向量中包含了大多数最近用过的虚拟页到物理页的映射（如图4-16所示）以及每页的页保护类型。TLB中每个项都类似一个高速缓存项，它的标识符保存了虚拟地址的一部分，它的数据部分则保存了一个物理页号、保护域、有效位，通常还有一个修改位，这些用来表明被高速缓存的页表项所对应页的状态。如果一个页表项的全局位被置为“1”（用于对所有进程都可见的系统空间页），当进程切换时TLB项仍然有效。

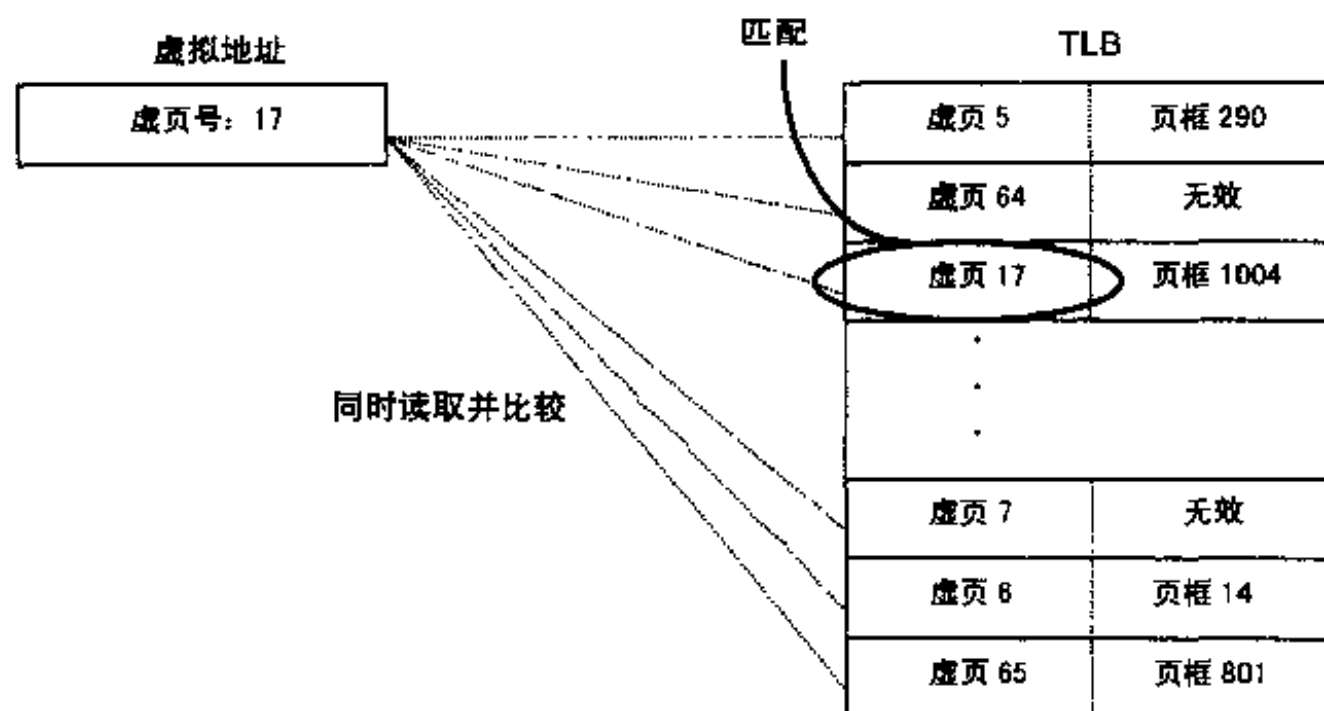


图4-16 访问快表

由于常用的虚拟地址很可能记录在TLB项中，这就使虚拟地址到物理地址的变换非常快，而且减少了对内存的访问。如果一个虚拟地址不在TLB中，它可能仍在内存中，但是需要对内存多次访问来找到它。如果一个虚页已经被调出了内存，或者内存管理器更改了页表项，内存管理器必须明确地将相应的TLB项置为无效。当进程再次访问这个虚页，就会产生缺页中断，内存管理器将该页再次调回内存，同时在TLB中为它重新创建项。

为了最大限度提高代码的通用性，只要可能，内存管理器对所有页表项均同等对待，无论它们是由硬件还是由软件维护。例如，当一个页表项由无效变为有效时，内存管理器会调用一个内核例程。此例程的任务是以体系结构所需要的方式将新的页表项装入TLB。在x86系统中，处理器装入TLB不需要软件干预，所以这部分代码是NOP。

4.2.3 用户空间内存分配方式

本节主要介绍Windows 2000/XP管理应用程序内存的方式。首先，介绍两个与内存分配相关的数据结构——虚拟地址描述符和区域对象。接着介绍三种管理应用程序内存的方法：

- 以页为单位的虚拟内存分配方法，适合于大型对象或结构数组；
- 内存映射文件方法，适合于大型数据流文件以及多个进程之间的数据共享；
- 内存堆方法，适合于大量的小型内存申请。

1. 虚拟地址描述符

内存管理器采用请求式页面调度算法将页面装入内存，即直到线程访问一个地址并引起缺页中断时，内存管理器才会执行页面调入操作。请求式页面调度是一种“懒惰计算”方式——直到一个任务被需求时它才被执行。

利用懒惰计算方法，分配连续的大块内存是一个很快的操作过程。然而，这种性能的提高是有代价的。因为只有线程实际访问内存时，内存管理器才会建立页表，所以内存管理器不能确定哪些虚拟地址是空闲的。为了解决这个问题，内存管理器维持了另一组数据结构来描述哪些虚拟地址已经在进程的地址空间中被保留，而哪些没有。这种数据结构被叫做“虚拟地址描述符”(virtual address descriptor, VAD)。对每个进程，内存管理器都维持有一组虚拟地址描述信息，用来描述进程地址空间的状态。虚拟地址描述信息被构造成一棵自平衡二叉树(self-balancing binary tree)以使查找更有效率。图4-17是一棵虚拟地址描述信息树。

当进程保留地址空间，或映射一个内存区域时，内存管理器创建一个VAD来保存分配请求所提供的信息，例如保留的地址范围，这个范围是共享的还是私有的，子进程是否能够继承这个地址范围的内容以及此地址范围内应用于页面的保护措施。

当线程首次访问一个地址，内存管理器必须为包含这个地址的页面创建一个页表项。为此，内存管理器找到一个包含被访问的地址的VAD，并利用所得的信息填充页表项。如果这个地址落在VAD覆盖的地址范围以外，或所在的地址范围仅被保留而未提交，内存管理器就会知道这个线程在试图使用内存前并没有分配内存，因此将产生一次访问违规。

2. 区域对象

“区域对象”(section object)在Win32子系统中被称之为“文件映射对象”，表示可以被两个或

更多进程所共享的内存块。区域对象也可以被映射到页文件或另一个外存上的文件。区域对象主要作用：

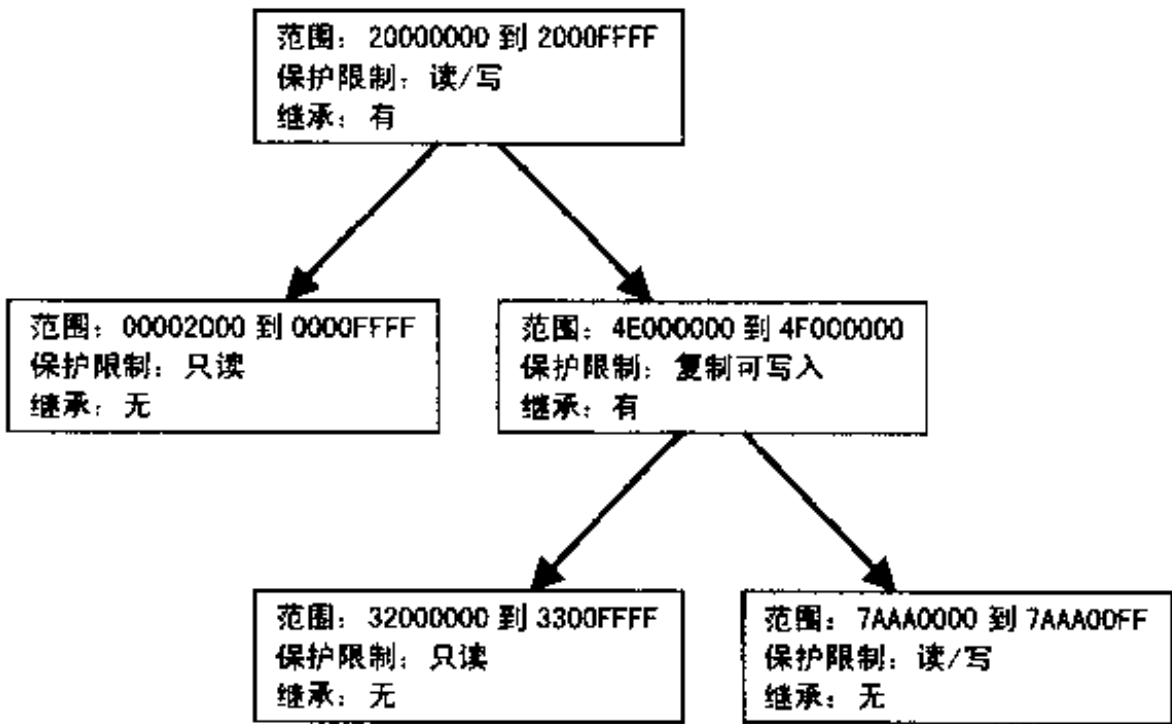


图4-17 虚拟地址描述信息树

- 系统利用区域对象将可执行的映像装入内存。
- 高速缓存管理器利用区域对象访问高速缓存文件中的数据。（后面会介绍有关高速缓存管理器是如何利用区域对象的。）
- 可以使用区域对象将一个文件映射到进程地址空间。然后访问这个文件就象访问内存中的一个大数据组，不是对文件进行读写。当程序访问一个无效的页面（不在物理内存中的页面）时，引起缺页中断，内存管理器会自动地将这个页面从映射文件调入内存。如果应用程序修改了页面，内存管理器在页面进行常规调度时将修改写回文件（或者应用程序可以通过利用Win32的FlushViewOfFile函数来刷新一个视图）。

对象类型	区域
对象体属性	最大规模 页保护限制 页文件/映射文件 基准的/非基准的
服务程序	创建区域 打开区域 扩展区域 映射/非映射视图 查询区域

图4-18 一个区域对象

图4-18显示了区域对象的结构。
表4-9总结了存储在区域对象中的特有属性。

表4-9 区域对象体属性

属 性	用 途
最大规模	区域可增长到的最大字节数；如果映射一个文件，则最大规模就是这个文件的大小
页保护限制	当创建区域时，分配给该区域的所有页面的基于页面的内存保护限制
页文件/映射文件	指出区域是否被创建为空（基于页文件），或是用于文件加载（基于映射文件）
基准的/非基准的	指出区域是为所有进程所共享相同的虚拟地址处的基准区域，还是可以出现在不同进程的不同虚拟地址处的非基准区域

图4-19显示了内存管理器维护的用于描述映射区的数据结构。这些数据结构确保从映射文件中读出的数据的一致性，而不管访问类型（打开文件、映射文件等等）。

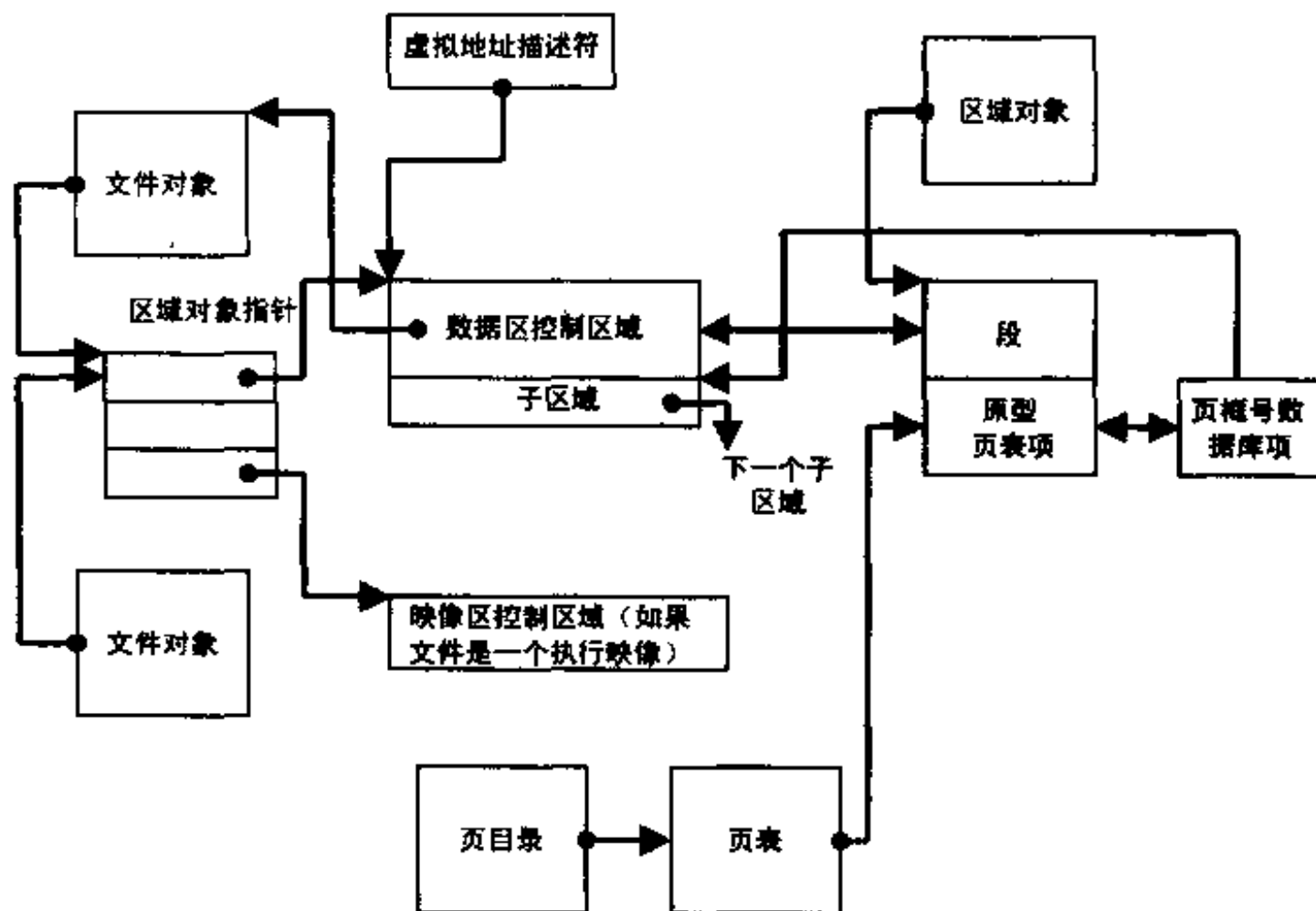


图4-19 内部区域结构

对于每个打开的文件（由一个文件对象表示），都有一个单独的区域对象指针结构。该结构为所有类型的文件访问维护数据一致性，同时也是为文件提供高速缓存的关键。区域对象指针结构由三个32位的指针组成：指向数据控制区域的指针、指向共享的高速缓存映射的指针和指向映像控制区域的指针。其中两个控制区域，分别被用来映射数据文件和可执行文件。

控制区域依次指向描述文件各个区域映射信息（只读、读写、写时复制等）的“子区域”结构。控制区域也指向一个在分页缓冲池中分配的“段”结构，这个段结构依次指向被用于映射到由区域对象的实际页面的原型页表项（缺页中断中会介绍）。进程页表指向这些原型页表项，它们依次映射正在被访问的页面。

3. 以页为单位的虚拟内存分配方式

在进程的地址空间中的页而或是空闲的(free)，或被保留(reserved)，或被提交(committed)。应用程序可以首先保留地址空间，然后向此地址空间提交物理页面。它们也可以通过一个函数调用同时实现保留和提交。这些功能是通过Win32 VirtualAlloc和VirtualAllocEx函数实现的。

保留地址空间是为线程将来使用所保留的一块虚拟地址。试图访问已保留内存会造成访问冲突，因为这时内存页面还没有映射到一个可以满足这次访问的存储器上。

在已保留的区域中，提交页面必须指出将物理存储器提交到何处以及提交多少。提交页面在访问时会转变为物理内存中的有效页面。提交的页面可以是私有的，也可以被映射到一个区域视

窗中（可能会，也可能不会被其他进程映射）。区域视图将在下面介绍。

如果页面是进程私有的并且以前从来没有被访问过，第一次被访问时它们被当做零初始化页创建。如果需要内存，私有的已提交页面可以自动被操作系统写入页文件。私有的提交页面对于其他进程是不可访问的，除非进程有使用交叉进程内存访问函数的权限，如ReadProcessMemory或WriteProcessMemory。提交的页面被映射到映射文件的某一部分时，除非当前进程已经访问过此页，或是另一个拥有相同映射文件的进程已经访问过，否则访问时需要从磁盘中读出。

可以通过VirtualFree或VirtualFreeEx函数回收页面或释放地址空间。回收和释放之间的区别与保留和提交之间的区别相似——回收的内存仍然被保留，但是释放的内存不被提交，也不被保留（它是空闲的）。

分两步保留和提交内存可以直到需要时才提交页面，这样减少了内存的使用。保留内存是Windows 2000/XP中既快速又便宜的操作，因为它不消耗任何物理页面（一种珍贵的系统资源）或进程页文件配额（进程可以消耗的提交页面数量的限制）。所需要更新或构造的是相对较小的代表进程地址空间状态的内部数据结构VAD。

地址空间被保留，然后当需要时再提交，而不是为全部区域提交页面，这对于需要潜在、大量和连续内存缓冲区的应用程序非常有用。在操作系统中这项技术被用于线程的用户态堆栈。当创建线程时，就保留一个堆栈（缺省值为1MB）。然而，只有两个页面被提交，一个在堆栈中用于初始时，另一个作为保护页捕获对超过堆栈提交部分的访问，在需要时，自动扩展堆栈。

4. 内存映射文件

内存映射文件可以用来保留一个地址区域，并将物理存储器提交给该区域。与虚拟内存的差别是：物理存储器来自位于磁盘上的文件。内存映射文件可以用于3个不同的目的：

- 加载和执行.exe和.dll文件，这可以节省应用程序启动所需的时间；
- 访问磁盘上的数据文件，这可以减少文件I/O，并且不必对文件进行缓存；
- 实现多个进程间的数据共享。

前面介绍的区域对象是系统实现这些功能的关键。区域对象在内存管理器中用来映射虚拟地址，无论它是在内存，还是在页文件，或是在其他文件中，应用程序访问它时，就象它在内存中一样。

区域对象可以连接到已打开的磁盘文件（映射文件），或是已提交的内存（提供共享内存）。可以调用Win32函数CreateFileMapping创建区域对象，其参数包括映射到区域对象的文件句柄（或是INVALID_HANDLE_VALUE表示页文件支持区域），还有可选的名字和安全描述符。如果区域有名字，其他进程可以用OpenFileMapping打开它。也可以通过句柄继承或句柄复制（通过使用DuplicateHandle）访问区域对象。设备驱动程序也可以使用ZwOpenSection，ZwMapViewOfSection，和ZwUnmapViewOfSection函数操纵区域对象。

区域对象可以指向那些比进程地址空间还大的文件。要访问一个非常大的区域对象，进程只能通过调用MapViewOfFile函数映射区域对象的一部分（叫做区域视图），并指定映射范围。由于在一个时刻只有需要的区域对象视图必须被映射到内存，进程可以只为映射视图保留地址空间。

Win32应用程序可以通过将文件映射到它的地址空间来方便地完成文件的I/O操作。不仅用户

应用程序使用区域对象，系统也通过区域对象加载可执行映像、动态链接库DLL以及设备驱动程序到内存；高速缓存管理器使用它们在缓存文件中存取数据。

像大多数的现代操作系统一样，Windows 2000/XP提供了在进程和操作系统之间共享内存的机制。共享内存可以定义为对于多个进程都是可见的内存，或者存在于多个进程的虚拟地址空间的内存。例如，如果两个进程使用同一DLL，这就意味着该DLL被访问的代码页面只装入物理内存一次，而其他映射此DLL的进程共享这些页面。如图4-20所示：

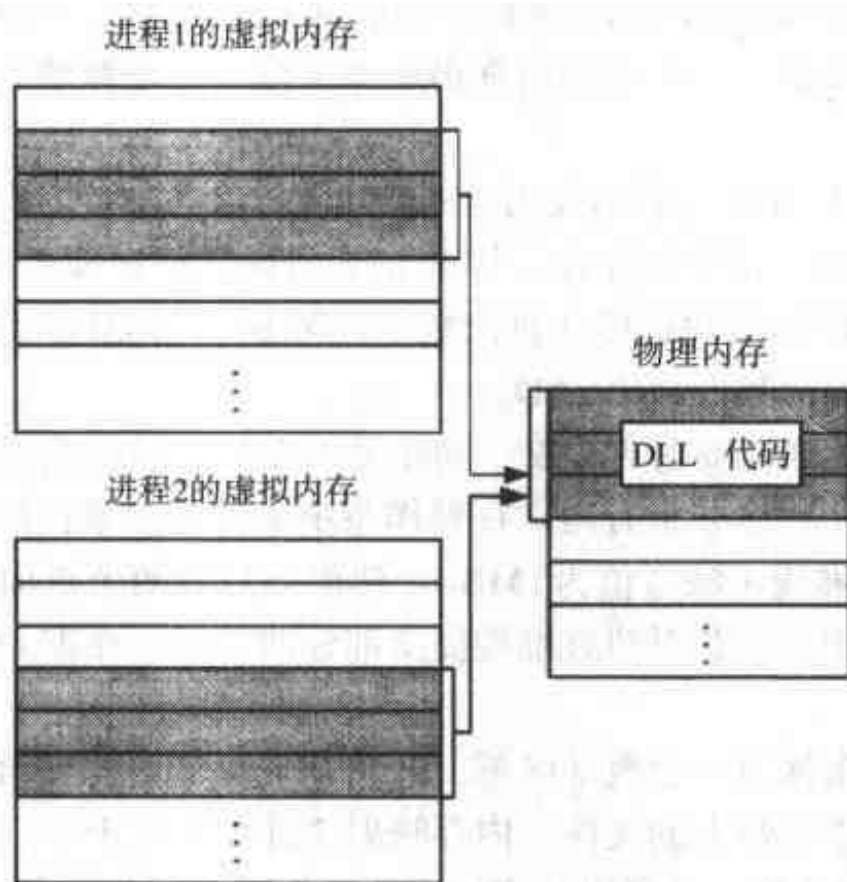


图4-20 进程间共享内存

5. 堆功能

堆(heap)是保留的地址空间中一个或多个页组成的区域，这个地址区域可以由堆管理器按更小块划分和分配。堆管理器是分配和回收可变内存的函数集（不必像VirtualAlloc一样按页对齐）。堆管理器的函数位于Ntdll.dll和Ntoskrnl.exe中。API子系统（如Win32的API）调用Ntdll中的函数，各种执行组件和设备驱动程序调用Ntoskrnl中的函数。

进程启动时带有一个缺省进程堆，通常是1MB大小（除非使用/HEAP链接标志在映像文件中指定其他值）。这个大小只是最初保留值，如果需要，它会自动扩大。（可以在映像文件中指定初始提交大小。）Win32的应用程序和一些需要分配临时内存块的Win32函数将使用这个进程缺省堆。进程也可以使用HeapCreate函数创建另外的私有堆。当进程不再需要私有堆时，可以通过调用HeapDestroy释放虚拟地址空间。只有用HeapCreate创建的私有堆（不是缺省堆）可以在一个进程的生命周期中被释放。

为了从缺省堆中分配内存，线程必须调用GetProcessHeap函数得到一个指向它的句柄。（这个函数返回描述该堆数据结构的地址，但调用程序一般不依赖它。）有了句柄后，线程可以调用HeapAlloc和HeapFree来从堆中分配和回收内存块。堆管理器还为每个堆提供了一个选项来串行

化分配和回收。这样多个线程可以同时调用堆函数，而不会破坏堆数据结构。缺省进程堆被缺省地设置了串行化选项。对于另外的私有堆，可以向HeapCreate传递一个标志来指定是否执行串行化功能。

堆管理器支持大量的内部认证检查，虽然目前这些检查没有正式文档化，但是可以在Windows 2000/XP支持工具、SDK平台和DDK中通过全程标志实用程序（Gflags.exe）在系统范围或单个映像基础上启动它们。许多标志对堆管理器的影响是自说明的。一般来说，激活这些标志后，如果有非法使用或破坏堆的操作，系统会通过异常或返回错误代码的方式将错误通知应用程序。

有关堆功能更多的信息，见Win32 API在MSDN的参考文件。

4.2.4 系统内存分配

内存管理器为设备驱动程序以及其他核心态组件提供了大量的服务，如分配和释放物理内存、锁定物理内存页面实现直接内存访问（DMA）等。另外，系统还提供了以Ex为前缀的例程，来分配和释放系统空间（分页和非分页缓冲池）。系统初始化时，内存管理器创建了两种动态大小的内存缓冲池，核心态组件可以用它来分配系统内存。

- **非分页缓冲池** 由长驻物理内存的系统虚拟地址区域组成，在任何时候，从任何IRQL级和任何进程上下文都可以访问。需要非分页缓冲池的原因是缺页中断在DPC/调度级（或更高级别）时不能被满足。
- **分页缓冲池** 在系统空间中可以被分页和换出的虚拟内存区域。那些不会从DPC/调度级（或更高级别）访问内存的设备驱动程序可以使用分页缓冲池。从任何进程上下文都可以访问它们。

两种内存缓冲池均位于系统空间，并被映射到每个进程的虚拟地址空间。（在表7-10，可以找到它们在系统内存的起始地址。）内核提供了ExAllocatePool等函数从这些缓冲池分配和回收内存。

系统有两种非分页缓冲池：一种在一般情况下使用，另一种小型的（4页）缓冲池在非分页缓冲池已满并且调用者不能允许分配失败时，紧急使用。单处理器系统有三个分页缓冲池；多处理器系统有五个。多个分页缓冲池减少了在同时调用缓冲池例程时，系统代码的阻塞频率。分页缓冲池和非分页缓冲池初始大小依赖于系统物理内存的大小，以后在需要时，可以一直增长到在系统引导时计算出的最大值。通过改变在注册表键HKLM\SYSTEM\CurrentControlSet\Control\SessionManager\MemoryManagement的Nonpaged PoolSize和PagedPoolSize值从0（此值引起系统重新计算大小）到期望的字节数，可以覆盖缓冲池的初始大小，但不能超过系统定义的缓冲池最大值。

系统计算出的大小被存储在4个内核变量中，其中三个可以通过性能计数器查询。这些变量和计数器，以及可以更改大小的两个注册表键都被列在表4-10中。

Windows 2000/XP同时也提供了一个快速内存分配机制，叫做后备链表(Look-Aside List)。缓冲池和后备链表之间的基本区别是，缓冲池分配的大小可以变化，而后备链表仅包含固定长度的块。虽然缓冲池分配能够提供比较灵活的方式，但是使用后备链表时系统不必查找适合分配大小的空闲内存，因而更快。

表4-10 系统缓冲池大小变量和性能计数器

内核变量	性能计数器	注册表键改变初始值	描 述
MmSizeOfNonPaged-PoolInBytes	Memory: Pool Nonpaged Bytes	不允许	当前非分页缓冲池大小
MmMaximumNon-PagedPoolInBytes	没有	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\NonPagedPoolSize	非分页缓冲池最大值
没有	Memory: Pool Paged Bytes	不允许	当前分页缓冲池虚拟大小
MmPagedPoolPage (页数量)	Memory: Pool Paged Resident Bytes	不允许	当前分页缓冲池物理 (常驻) 大小
MmSizeOfPagedPool-InBytes	没有	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagedPoolSize	分页缓冲池最大值 (虚拟)

执行程序组件和设备驱动程序可以使用ExInitializeNpagedLookasideList和ExInitializePagedLookasideList函数，按照频繁分配的数据结构大小创建后备链表（在DDK中有说明）。为了减少多处理器同步的开销，几个执行程序子系统（像I/O管理器，缓存管理器和对象管理器）分别为每个处理器频繁访问的数据结构创建了后备链表。执行程序还针对小量内存分配（256字节或更少）为每个处理器创建了分页和未分页后备链表。

如果一个后备链表为空（例如第一次创建时），系统必须从分页或非分页缓冲池进行分配。如果有空闲块，这个分配将很快被满足。当块被返回时，链表增长。缓冲池例程根据设备驱动程序或子系统从链表分配内存的频繁程度来自动调整后备链表的空闲缓冲区数量，分配越频繁，更多缓冲区存入链表。如果后备链表不经常分配，它的大小会自动减少。（当平衡集管理器系统线程被唤醒并调用KiAdjustLookasideDepth函数时，这种检查每秒钟发生一次。）

4.2.5 缺页处理

前面介绍了当页表项有效时如何解决地址变换问题。当页表项的“有效”标志位被清除时，就表示所需的页由于某种原因对当前进程是不可访问的。这节将要讲述各类无效的页表项，以及对它们的访问是如何实现的。

对无效页面的一次访问称为“缺页错误”。内核中断处理程序将这类错误调派给内存管理故障处理程序（MmAccessFault）来解决。这个例程运行在引起错误的线程环境下，并负责尝试解决这个错误（如果可能）或引发适当的异常。这些错误可能由列举在表4-11中的多种情况引发。

表4-11 访问错误的原因

错误原因	结 果
所访问的页面没有驻留在内存，而是在磁盘上某个页文件或映射文件中	分配一个物理页面，将所需的页从外存读出，并放入工作集
所访问的页面在后备链表或修改链表中	将此页迁移到进程或系统工作集
所访问的页面未被提交（比如，保留的地址空间或未被分配的地址空间）	访问违规(access violation)
从用户态访问一个只能在核心态下访问的页面	访问违规
对一个只读页面执行写操作	访问违规
访问一个请求零页	给进程工作集添加一个由零初始化的页
对一个写保护页面执行写操作	写保护页违规（如果是访问一个用户态堆栈，则执行自动堆栈扩充）
对一个写时复制的页面执行写操作	制作进程私有（或会话私有）的页面拷贝，并置换原先的进程、会话，或系统工作集
所访问的页面在系统空间，它虽然是有效的，但没被包含在进程页目录中（例如，如果分页缓冲池在进程页目录创建后扩展）	从主系统页目录结构复制页目录项，并消除异常
在一个多处理器系统中，对一个有效但尚未执行写操作的页执行写操作	在页表项中将修改位置“1”

下面介绍由访问错误处理程序处理的四个基本类型的无效页表项。再下面的一节是对一个无效页表项的特例——原型页表项——进行解释，它主要用于实现页面共享。

1. 无效的页表项

下面详述四种类型的无效页表项(Invalid PTE)及其各自的结构。其中有些标识位与表4-8所描述的硬件页表项标识位相同。

- **页文件（page file）** 所需的页驻留在一个页文件中，并引发页面调入操作，如下图4-21所示：

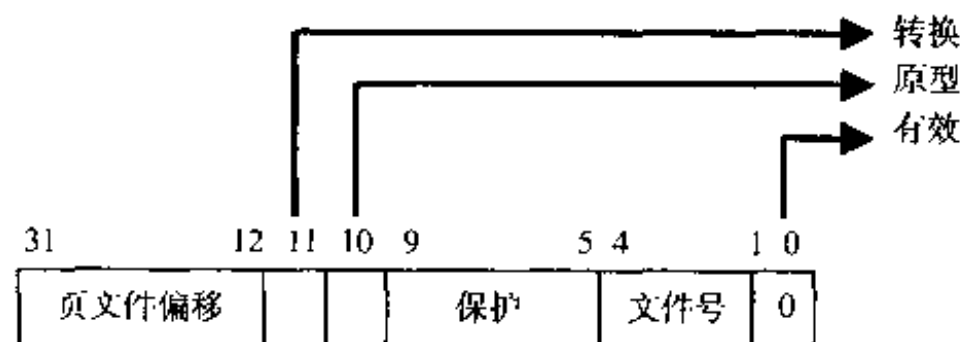


图4-21 页文件无效页表项

- **请求零页（demand zero）** 所需的页必须是零页面。页面调度程序查看零页链表。如果这个链表为空，页面调度程序从空闲链表中取出一页并把它置零。如果空闲链表也为空，页面调度程序则从后备链表中取出一页并把它置零。此类页表项的格式与上面所示的页文件的页表项格式相同，但是页文件号和页文件偏移均为零。
- **转换（transition）** 所需页面在内存中的后备链表、修改链表或修改尚未写入链表。从链表中删除此页，并添加到工作集，如图4-22所示：

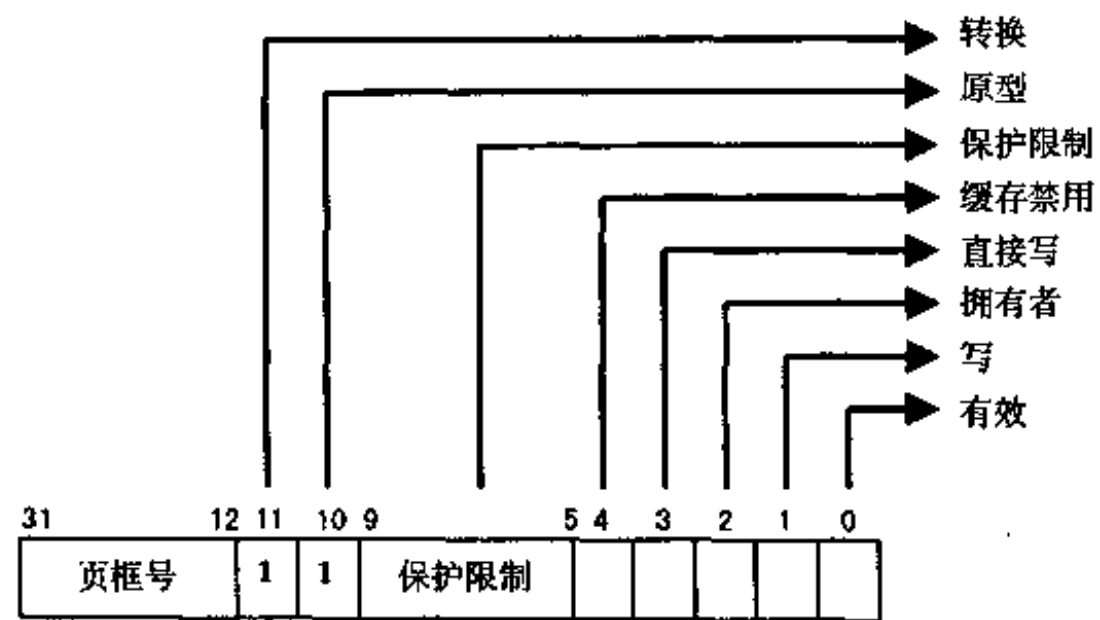


图4-22 转换无效页表项

• **未知 (unknown)** 页表项为零，或者页表不存在。出现这两种情况，就意味着应该检查虚拟地址描述信息 (VAD)，以确定这个虚拟地址是否已经被提交。如果已经提交，则建立页表来表示新近提交的地址空间。

2. 原型页表项

如果一个页面可能被两个进程所共享，内存管理器依靠一个称为“原型页表项” (prototype PTE，原型PTE) 的软件结构来映射这些潜在的共享页面。当一个区域对象第一次被创建时，一系列原型页表项同时被创建。

当进程首次访问一个映射到区域对象视图的页面时，内存管理器利用原型页表项中的信息，填入进程页表中用于地址变换的实际页表项。当共享页面为有效时，进程页表项和原型页表项均指向包含数据的物理页。为了追踪访问有效共享页面的进程页表项数量，“页框号数据库项”内增加了一个计数器。这样，内存管理器就能确定一个共享页面何时已经不再被任何页表引用，这个页面就可以被标记为无效，并移到转换链表或写回外存。

使一个共享页面无效时，进程页表中的页表项由一个特殊的页表项来填充。这个特殊的页表项指向描述该页面的原型页表项，如图4-23所示：

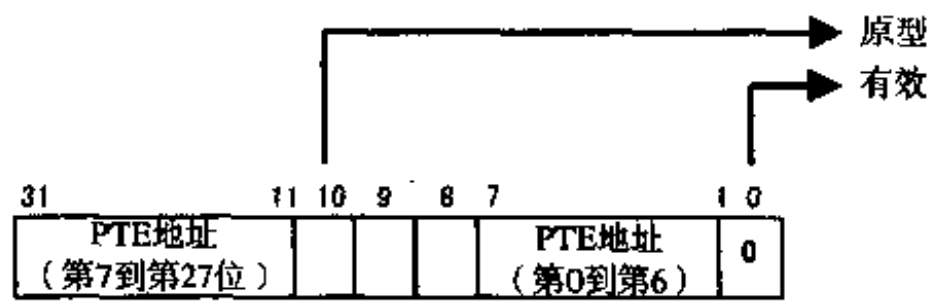


图4-23 一个指向原型页表项的无效页表项结构

这样，当该页以后被访问时，内存管理器就可以利用这个页表项中编码信息定位原型页表项，而原型页表项描述被访问的页面。共享页面可以是原型页表项所描述的6种状态中之一：

- **活动/有效 (active/valid)** 由于另一个进程曾经访问过此页面，页面保存在物理内存中。
- **转换** 所需页面在内存的后备链表或修改链表中。

- 修改尚未写入 (modified-no-write) 所需页面在内存的修改尚未写入链表中。
- 请求零页 所需页面应是一个零页面。
- 页文件 所需页面驻留在页文件中。
- 映射文件(mapped file) 所需页面驻留在映射文件中。

虽然这些原型页表项的格式与前面所述的实际页表项的格式相同,但是这些原型页表项并不用于地址变换——它们是介于页表和页框号数据库之间的一层,决不会直接出现在页表中。

通过将潜在共享页面的所有访问程序指向原型页表项来解决缺页问题,内存管理器可以方便地管理共享页面,而无须更新每个共享此页面进程的页表。例如,一段共享代码或者一个数据页面可能在某个时候被调出到外存。当内存管理器将此页重新从外存调进时,只需更新原型页表项,使之指向此页新的物理位置——而共享此页的诸进程的页表项始终不变(清除有效位,且仍指向原型页表项)。此后,当进程访问该页时,实际的页表项才得到更新。

图4-24表示映射视图中的两个虚页。一个是有效的,另一个无效。如图所示,第一页是有效的,并且进程页表项和原型页表项均指向它。而第二页在页文件中——原型页表项保存着它的确切位置。进程(以及其他映射到该页的进程)的页表项指向这个原型页表项。

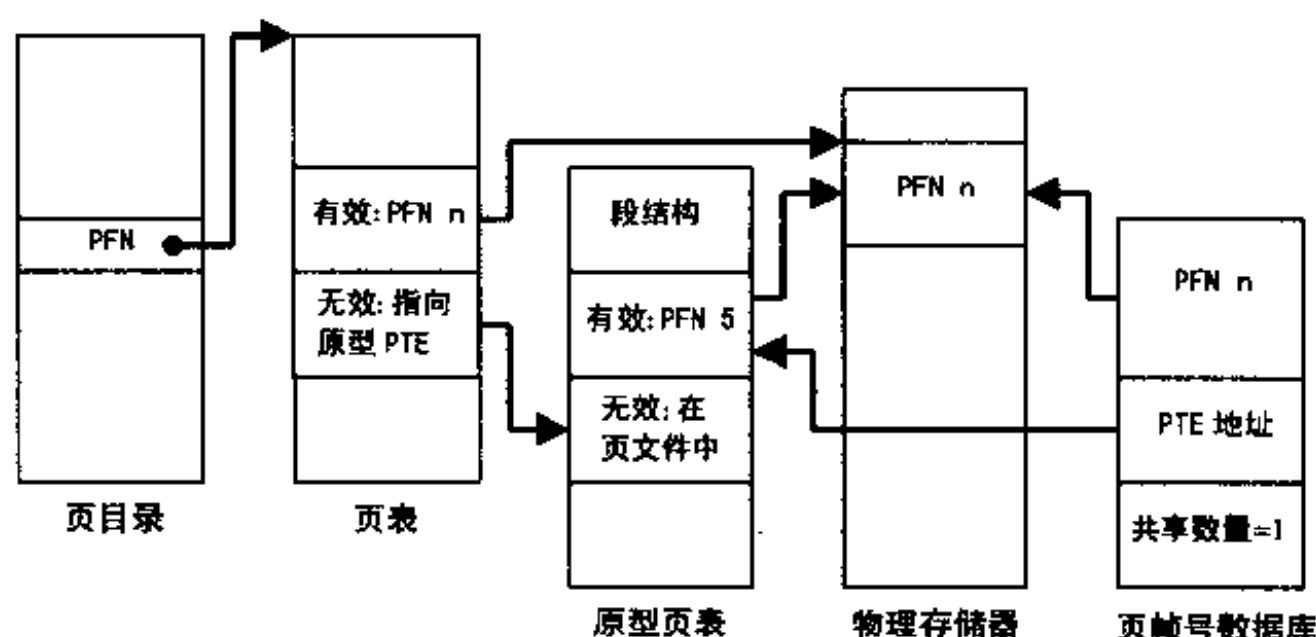


图4-24 原型页表项

3. 页面调入I/O

当必须向文件(页或映射文件)发出读操作未解决缺页问题时,将产生页面调入I/O。同样,因为页表也是可分页的(可以被换出),所以当系统装载包含页表项或描述被访问的原始页面的原型页表项的页表页时,缺页错误处理还可能导致其他的缺页错误。

页面调入I/O操作是同步的——即线程一直等待I/O完成——并且是不能被异步过程调用(APC)中断。页面调度程序使用I/O请求功能中一个特别的修改程序来执行调页I/O。调页I/O完成时,I/O系统触发一个事件来唤醒页面调度程序,并允许它继续页面调入处理。

当进行调页I/O操作时,缺页线程没有临界内存管理同步对象。进程中的其他线程仍可以使用虚拟存储器函数,并在调页I/O发生时处理缺页错误。因此,在I/O结束时页面调度程序必须识别下面这些情况:

- 同一进程中的另一线程，或一个其他的进程，都可能由于一个相同的页面导致缺页错误。（称为“冲突页错误”，将在下节介绍。）
- 页面可能已经从虚拟地址空间中被删除（并重新映射）。
- 页面的保护限制可能发生了变化。
- 错误可能是由一个原型页表项引发的，并且这个原型页表项所映射的页面可能并不在工作集中。

页面调度程序通过在提出调页I/O请求之前，在线程的核心堆栈中保存足够多的状态信息来处理这些情况，这样当请求结束时，页面调度程序就能检测到这些情况。如果需要，可以不把页面置为有效来消除这次缺页错误。当错误指令再次发出时，页面调度程序会再次被调用，而页表项也会在新状态下被再次评估。

4. 冲突页错误

另一个线程或进程也对正在被调入的页面产生缺页错误，这种情况称为“冲突页错误”（collided page fault）。由于它们通常发生在多线程系统中，页面调度程序能够检测，并很好地处理冲突页错误。

当页面调度程序检测到冲突页错误时，会通告此页面在转换中并且正在读取。（这个信息保存在页框号数据库项中。）在这种情况下，页面调度程序对页框号数据库项中的特定事件发出一个等待操作。该事件是由首次发出解决缺页错误I/O的线程来初始化。

I/O操作完成后，所有等待该事件的线程都会被唤醒。第一个获得页框号数据库锁的线程负责执行完成页面调入的操作。这些操作包括为确保I/O操作顺利完成面对I/O状况的检测、清除页框号数据库中的“正在读入位”以及对页表项的更新。

当后续的线程获得页框号数据库锁来完成冲突页错误时，页面调度程序确认，正在读入位已清零时初始更新已经完成；接着检测页框号数据库项中的页面调入错误标识位以保证页面调入I/O成功完成。如果页面调入错误标识位被置“1”，页表项将不会更新，并且在出错线程中产生一个页面调入错误异常。

5. 页文件

现代操作系统能够使磁盘空间看起来像内存一样，为进程提供了虚拟存储器。Windows 2000/XP中磁盘上的部分通常称为“页文件”。如果计算机有64 MB物理内存，同时在磁盘上有100 MB的页文件，那么应用程序就可以认为计算机总共拥有164 MB内存。

性能计数器中的Process: Page File Bytes实际上就是被提交的进程私有内存总和。这些内存可能有一些，或全部在页文件中，也可能全都不在页文件中。内存管理器一直在追踪已提交私有内存的使用情况，该情况对整个系统而言称为“提交量”，对各个进程而言称为“页文件限额”。（此外，内存使用情况并不反映页文件使用情况——它只反映提交的私有内存使用情况。）无论何时向虚拟地址提交物理页面，都会查询提交量和页文件限额。一旦达到系统整体提交限制（物理内存和页文件都满了），这次虚存分配就会失败，直到一些进程释放内存（比如，当一个进程撤销）。

Windows 2000/XP支持最多16个页文件。当系统启动时，会话管理器进程读取页文件链表，并

通过检查注册值HKLM\SYSTEM\CurrentControlSet\Control\SessionManager\MemoryManagement\PagingFiles打开页文件。如果没有指定的页文件，系统在引导区创建一个缺省的20 MB页文件。（嵌入式版本，如嵌入式的Windows NT 4，没有缺省的页文件。）一旦打开页文件，在系统运行期间不能删除，因为系统进程为每个页文件都维持一个打开的句柄。

要增加一个新的页文件，控制面板可以使用在Ntdll.dll中定义的“NtCreatePagingfile”系统服务程序。即使页文件所在的目录是压缩的，它也以非压缩的形式被创建。为了避免新的页文件被删除，一个句柄会被复制到系统进程，这样当创建进程关闭这个新页文件的句柄时，其他进程仍可打开这个页文件。

表4-12列出性能计数器可以基于系统范围或基于每个页文件的范围，来检查被提交的私有内存使用状况。但无法确切知道一个进程被提交的私有内存中有多少常驻内存，有多少被调出到页文件中。

表4-12 被提交的内存和页文件性能计数器

性能计数器	说 明
Memory: Committed Bytes	已提交的虚拟（不是保留的）内存字节数，该数字不一定表示页文件的使用情况，因为它包括物理内存中提交的私有页面，而这部分页面是不会被调出的。更合适的说法是，当进程完全不驻留内存时，它表示能够使用的页文件空间总量
Memory: Committed Limit	无需扩展页文件就能被提交的虚拟内存字节数；如果页文件能扩展，这个限界可以改变
Paging File: %Usage	页文件提交的百分比
Paging File: %Usage Peak	页文件提交的最高百分比

4.2.6 工作集

现代计算机系统中内存的访问速度远远高于外存的访问速度。如果系统中不产生缺页中断，则访问数据的时间约等于内存访问时间。但是如果发生缺页中断，则需要从外存中将该页调入，因此会大大降低系统性能。通过对缺页率的长期研究，Denning提出了工作集理论。由于程序在运行时对页面的访问是不均匀的（即局部性），如果能够预知程序在某段时间内要访问那些页面，并将它们提前调入内存，这将降低缺页率，提高CPU利用率。

本章中用来描述驻留在物理内存中的虚拟页面子集的术语叫做“工作集”。在过去的几节里，我们已经集中介绍了Windows 2000/XP进程的地址空间布局、地址变换过程、页表以及页表项等。下面我们将解释Windows 2000/XP是如何根据局部性原理，在物理内存中保持一个虚拟地址的子集来提高效率。工作集分为两种——进程工作集和系统工作集。

注 为了支持Windows 2000/XP终端服务程序（它在一个单独的Windows 2000/XP Server系统中支持多个独立的、可交互的用户会话程序）所进行的内核扩展，添加了一个第三类工作集：会话工作集。

在分析每种工作集的细节之前，首先需要了解决定哪些页面被调入物理内存以及它们保留多长时间的总体策略。然后，再研究上述两种工作集。

1. 页面调度策略

Windows 2000/XP的内存管理器利用请求式页面调度算法以及簇方式将页面装入内存。当线程产生一次缺页中断时，内存管理器将引发中断的页面及其后续的少量页面装入内存。这个策略试图减少线程引起的调页I/O数量。因为根据局部性原理，程序，尤其是大程序，往往在一段特定的时间内仅在它地址空间中的一小块区域上运行，装入虚拟页面簇就减少了读取外存的次数。缺省页面读取簇的规模大小取决于物理内存的大小，详见表4-13。注意代码页相对于其他页面在这个数值上的差别。

表4-13 缺页故障读取簇的大小

内存大小*	代码页面的簇 规模的映像数	数据页面的簇 规模的映像数	其他页面的簇 规模的映像数
< 12 MB	3	2	5
12-19 MB	3	2	5
> 19 MB	8	4	8

* Windows 2000/XP支持的最小的内存规模是32 MB。但是，今后的嵌入式版本可以支持更小内存的系统。

当线程产生缺页中断时，内存管理器还必须确定将调入的虚拟页放在物理内存的何处。用于确定最佳位置的一组规则称为“置页策略”。选择页框应使CPU内存高速缓存不必要的震荡最小。因此Windows 2000/XP 需要考虑CPU内存高速缓存的大小。

如果缺页错误发生时物理内存已满，“置换策略”被用于确定哪个虚页面必须被从内存中移出为新的页面腾出空位。在多处理器系统中，Windows 2000/XP采用了局部先进先出置换策略。而在单处理器系统中，Windows 2000/XP的实现更接近于最近最久未使用策略（LRU）（称为“时钟算法”，用于大多数版本的UNIX）。Windows 2000/XP为每个进程分配一定数量的页框（动态调整），称为“进程工作集”（或者为可分页的系统代码和数据分配的页框，称为“系统工作集”）。当进程工作集达到它的限界，或者由于有其他进程对物理内存的请求而需要对工作集进行修剪时，内存管理器只好从工作集中移出页面，直到它确认有足够的空闲页为止。下面介绍如何管理工作集。

2. 工作集管理

在开始时所有进程缺省的工作集最大值和最小值是相同的。在系统初始化时基于物理内存的大小计算出这些数值，列于表4-14。

表4-14 缺省的最大和最小工作集的大小

内存规模	缺省的最小工作集大小（页）	缺省的最大工作集大小（页）
小（小于19 MB）	20	45
中（20 ~ 32 MB）	30	145
大（Windows 2000/XP Professional大于32 MB， Windows 2000/XP Server大于64 MB）	50	345

每个进程可以利用Win32函数SetProcessWorkingSet来更改这些缺省值，但必须拥有“增大调度优先级”的用户权限。工作集的最大规模不能超过系统初始化时计算出的并保存在内核变量MmMaximumWorkingSetSize中的最大值。

当缺页错误发生时，先要检测进程的工作集限制和系统中空闲内存的数量。如果情况允许，内存管理器允许进程把工作集规模增加到最大值（如果有足够的空闲页，也可以超过这个最大值）。然而，如果内存紧张，缺页错误发生时Windows 2000/XP则替换而不是增加工作集中的页面。

当频繁地发生页面修改，或需要更多的内存来满足内存需求时，Windows 2000/XP可以通过将修改过的页面写回外存来保持更多可用内存。因此，当物理内存变得很低时（MmAvailablePages少于MmMinimumFreePages），“工作集管理器”自动修剪工作集，以增加系统中可用的空闲内存数量。工作集管理器是运行在平衡集管理器系统线程环境下的一个例程。（利用前面提到的Win32函数SetProcessWorkingSet，也可以在应用程序初始化后修剪自己进程的工作集。）

工作集管理器检测可用内存，并决定哪个工作集需要被修剪。如果有充足的内存，工作集管理器将计算有多少页面而需要从工作集中被移出。如果修剪是必须的，它选择大于其最小值的工作集。它也会动态调整检查工作集，并按优先顺序排列候选的待修剪进程链表。例如，已经等待较长时间的大进程比频繁运行的小进程优先修剪；前台运行的应用程序应最后修剪等等。

影响工作集扩展和修剪的一些内核变量列于表4-15，这些变量的值是确定的，或由系统设置的，不能被注册值调整。

表4-15 与工作集有关的系统控制变量

变 量	值	描 述
MmWorkingSetSize_Increment	6	有足够的可利用页面并且工作集小于其最大值时，添加到工作集的页面数量
MmWorkingSetSize_Expansion	20	工作集达到最大值并且有足够的可利用页面时，用于扩展工作集的页面数量
MmWsExpandThreshold	90	扩展工作集超出其最大值时必须拥有的可使用页面数量
MmPagesAboveWs_Minimum	动态	每个工作集均为其最小值时，将从工作集中移出的页面数量
MmPagesAboveWs_Threshold	37	当内存短缺且变量MmPagesAboveWsMinimum的值大于该值，修剪工作集
MmWsAdjustThreshold	45	在试图缩小工作集之前，工作集缩小需要释放的页面数量
MmWsTrimReduction_Goal	29	工作集修剪释放的页面总量

3. 平衡集管理器和交换程序

工作集扩展和修剪发生在一个称作“平衡集管理器”（例程KeBalanceSetManager）的系统线程环境下。平衡集管理器是在系统初始化时被创建的。虽然从技术的角度看，平衡集管理器是内核的一部分，但它通过调用内存管理器的工作集管理器来完成工作集的分析 and 调整。

平衡集管理器等待两个不同的事件对象：在每秒激发一次的周期计时器到期后产生一个事件；另一个是内部工作集管理器事件，由内存管理器确定工作集需要调整时，在不同时候发出。

例如，如果系统目前缺页错误率很高，或者空闲链表太小，内存管理器就会唤醒平衡集管理器，这样它将调用工作集管理器开始修剪工作集。如果内存充足，工作集管理器通过把所缺页面调入内存的方式增加进程的工作集，但是工作集只能根据需要增加。

当平衡集管理器由自身的1秒计时器到期而被唤醒时，会经历下面4个步骤：

1) 平衡集管理器每被唤醒4次就会将产生一个事件。这个事件唤醒另一个系统线程，称为交换程序（KeSwapProcessOrStack例程）。

2) 平衡集管理器检查后备链表，如果必要还将调整它们的深度（为了改善访问时间，并减少缓冲池的碎片）。

3) 平衡集管理器寻找处于CPU“饥饿状态”而需要提高其优先级的线程。

4) 平衡集管理器调用内存管理器的工作集管理器。（工作集管理器有它自己的内部计数器，用来调节何时执行工作集修剪和如何迅速修剪。）

如果需要运行的线程内核堆栈被换出内存，或者进程已经被换出内存，交换程序也可以由内核的调度代码唤醒。交换程序寻找在一段时间内一直处于等待状态的线程（小内存系统3秒，中内存或大内存系统7秒）。如果找到一个，则将线程的内核堆栈转移（将页面移入修改链表或后备链表），以收回它的物理内存。操作所遵循的原则是：如果一个线程已经等待了相当长的时间，那么它将等待更长时间。当进程最后一个线程的内核堆栈也从内存中移去时，这个进程将标记为被完全换出。这也是已经等待很长时间的进程（如Winlogon）可以有零工作集的原因。

4. 系统工作集

正如进程拥有工作集一样，操作系统中可分页的代码和数据由一个单独的“系统工作集”管理。系统工作集中可以驻留5种不同的页面：

- 系统高速缓存页面。
- 分页缓冲池。
- Ntoskrnl.exe中可分页的代码和数据。
- 设备驱动程序中可分页的代码和数据。
- 系统映射视图（部分映射在0xA0000000处，如Win32k.sys）。

可以利用表4-16所示的性能计数器或系统变量来考察系统工作集的大小或者影响系统工作集5个部分的大小。注意，性能计数器的数值是以字节为单位，而系统变量则是以页面数来度量。

表4-16 系统工作集性能计数器

性能计数器（字节）	系统变量（页）	描 述
Memory: Cache Bytes*	MmSystemCacheWs.Work_IngSetSize	系统工作集的总和（包括高速缓存、分页缓冲池、可分页的Ntoskrnl和驱动程序代码以及系统映射视图），而不仅仅是系统高速缓存的大小
Memory: Cache Bytes Peak	MmSystemCacheWs.Peak	系统工作集大小的峰值
Memory: System Cache Resident Bytes	MmSystemCachePage	系统高速缓存占用的物理内存

(续)

性能计数器(字节)	系统变量(页)	描 述
Memory: System Code Resident Bytes	MmSystemCodePage	Ntoskrnl.exe中可分页代码占用的物理内存
Memory: System Driver Resident Bytes	MmSystemDriverPage	可分页设备驱动程序代码占用的物理内存
Memory: Pool Paged Resident Bytes	MmPagedPoolPage	分页缓冲池占用的物理内存

* 在内部,该工作集被称为系统“高速缓存”工作集,尽管系统高速缓存仅仅是组成这个工作集的5种不同成分之一。因此,当它们显示系统工作集的总大小时,一些实用程序则认为它们仅显示了文件高速缓存的大小。

系统工作集的最大和最小值基于计算机物理内存的数量和系统是Windows 2000/XP Professional还是Windows 2000/XP Server,是在系统初始化时计算出来的。表4-17的初始值是基于系统内存大小选取的。

表4-17 系统工作集的最小值和最大值

内存规模	系统工作集最小值(页)	系统工作集最大值(页)
小	388	500
中	688	1150
大	1188	2050

4.2.7 物理内存管理

工作集描述了进程或系统拥有的驻留页面,而“页框号(PFN)数据库”描述了物理内存中各个页面的状态。页面一定处于表4-18所示的8种状态之一。

表4-18 页面状态

状 态	描 述
活动(又称有效)	页面是工作集的一部分(或者是进程工作集,或者是系统工作集)或根本不属于任何工作集(比如非分页的内核页面),并且有一个有效的页表项指向它
过渡(transition)	不在工作集中且不属于页面调度链表的页面暂时状态。当对该页面的I/O正在进行时,页面就处于这个状态。为了正确识别和处理冲突错误,需要对页表项进行编码
后备(stand by)	以前属于工作集但已被删除的页面。页面在写入磁盘后就未被修改过。页表项仍然指向这个物理页,但被标记为无效和处于过渡状态
修改	以前属于工作集但已被删除的页面。然而,页面在使用过程中修改过,并且它当前的内容未写入磁盘。页表项仍然指向这个物理页,但是被标记为无效和处于过渡状态。该页面被重新使用之前必须先写入磁盘
修改不写入	与修改页面相同,但它已经被标记,以使内存管理器的修改页面写回器不会将页面写入磁盘。在文件系统驱动程序发出请求时,高速缓存管理器标记页面为修改不写入

(续)

状 态	描 述
空闲	页是空闲的，但有不确定的数据（由于安全原因，这些页在用零初始化前不能当做用户页交给用户进程使用）
零初始化(zeroed)	页是空闲的，并且已经由零页初始化线程初始化
坏	页面已经产生了奇偶校验或者其他硬件错误，不可用

页框号数据库由描述系统中内存的各个物理页面的结构数组组成。页框号数据库及它与页表的联系如图4-25所示。有效的页表项指向页框号数据库中的项，且页框号数据库项（对于非原型页框号）指回利用它们的页表。至于原型页框号，它们指回原型页表项。

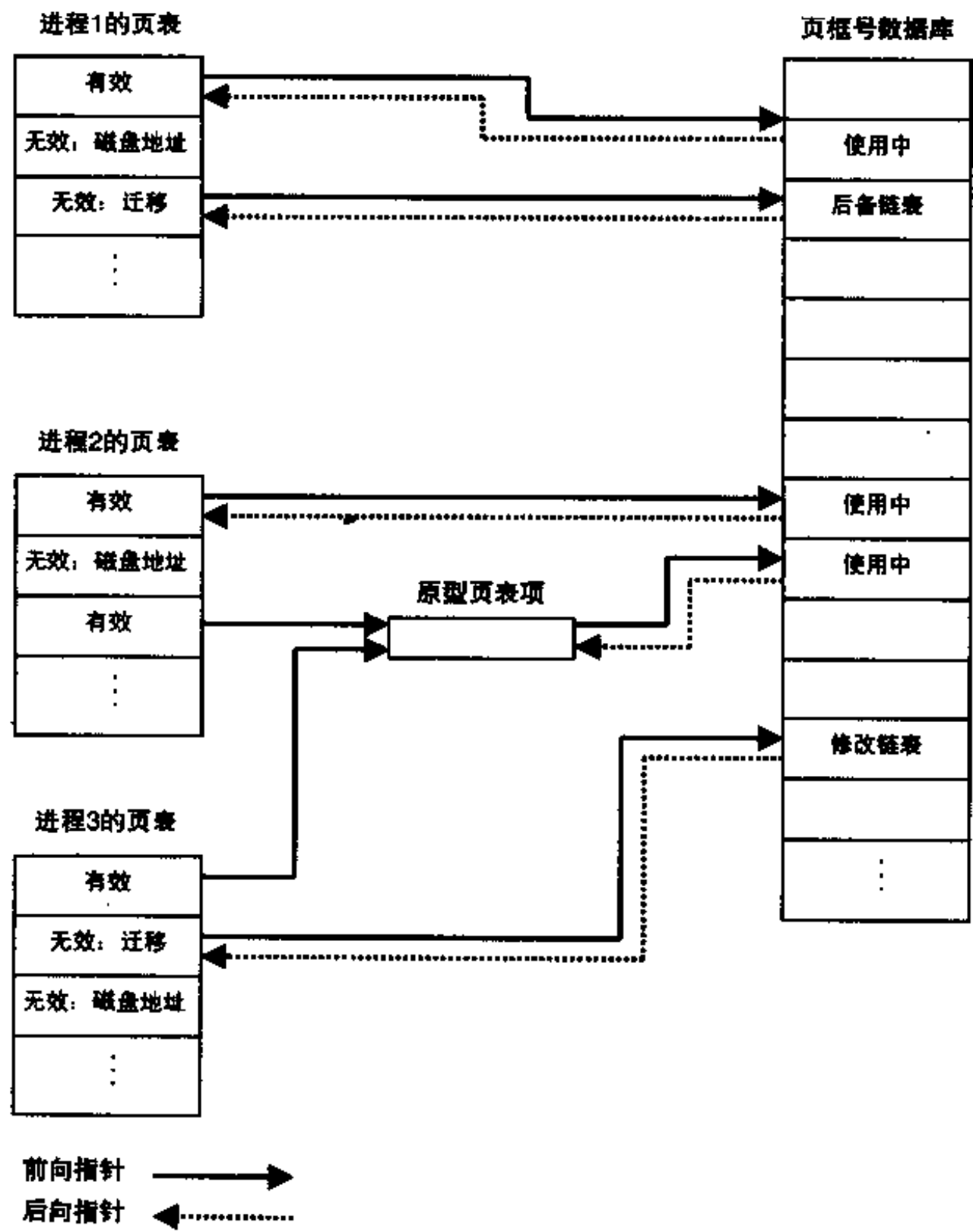


图4-25 页表与页框号数据库

在表4-18列出页面状态中，有6种组成了链表，以便内存管理器可以很快定位某特定类型的页面。（活动/有效页面和转换页面不在系统范围的页链表中。）图4-26显示了这些项是如何被链接在一起。

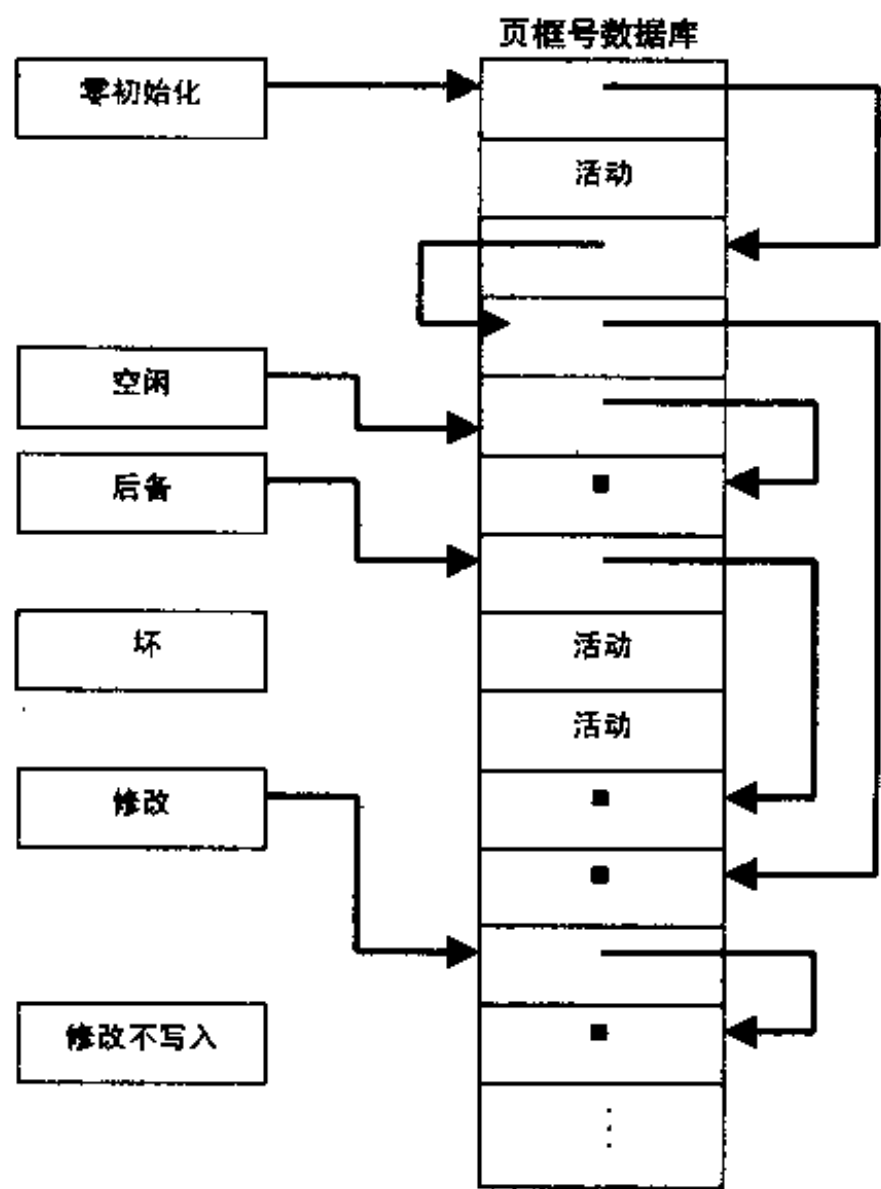


图4-26 页框号数据库中的页链表

下一节，将介绍这些链表如何用于处理缺页错误，以及页面如何移入和移出这些不同的链表。

1. 动态页链表

图4-27是一个页框迁移的状态框图。为了简单起见，修改不写入的链表没有被列出。

页框以如下步骤在页链表间移动：

- 当内存管理器需要一个零初始化的页面来满足一次请求零页缺页错误时，它首先试图从零页链表中得到一个页面。如果这个链表为空，则从空闲链表中选取一页并将其用零初始化。如果空闲链表也为空，则改从后备链表中选取一页并将其用零初始化。

需要零初始化页面的原因之一是满足C2级安全需求（C2 security requirement）。C2级要求必须分配给用户态进程初始化过的页框，以防止它们读取以前进程内存中的内容。因此，内存管理器提供用户态进程零初始化过的页框，除非读入的页面来自映射文件。如果来自映射文件，内存管理器可以使用非零初始化的页框，而用磁盘上的数据来初始化它们。

零页链表是由一个称为“零页线程”（系统进程中的线程0）的系统线程从空闲链表中移过来

的。零页线程等待一个事件对象来通知它开始工作。当空闲链表有8个或更多的页时，这个事件被激活。但是，零页线程仅在没有其他线程运行时才会运行，因为零页线程的优先级为0，而用户线程可以设置的最低优先级为1。

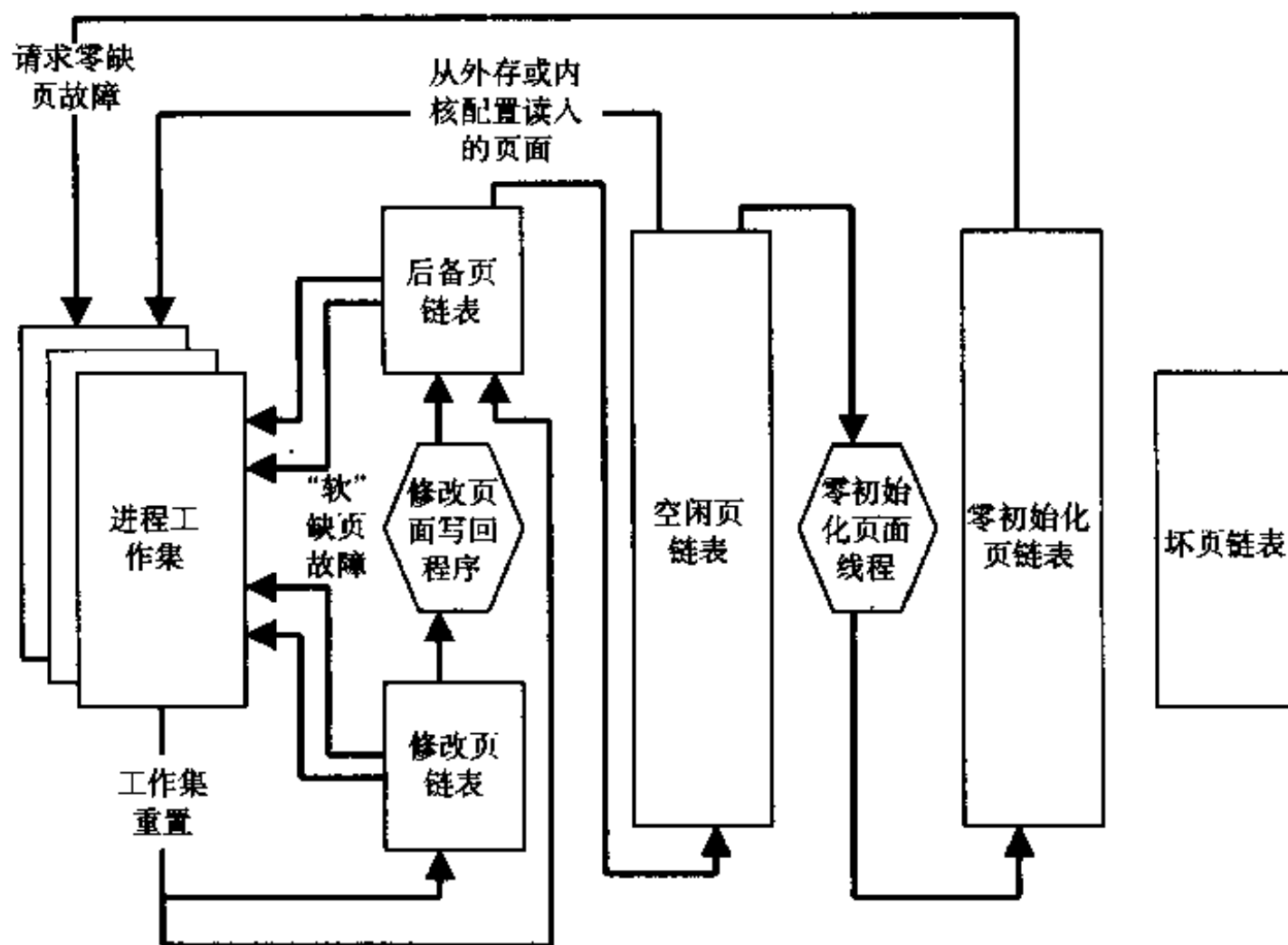


图4-27 页框的状态图

- 当内存管理器不需要零初始化页面时，它首先访问空闲链表；如果为空，则访问后备链表。在内存管理器使用一个来自后备链表的页框之前，它必须首先回溯，并清除无效页表项（或原型页表项）中仍指向这个页框的索引。由于页框号数据库中的项包含指向用户页表项（或共享页面的原型页表项）的指针，内存管理器可以快速找到该页表项并作适当的改变。
- 当进程不得不从工作集中放弃一个页面时（或者因为访问一个新页面而工作集已满，或者因为内存管理器修剪工作集），如果页面是干净的（未修改过），该页将加入后备链表；如果页面在驻留时修改过，则加入修改链表。当进程撤销时，所有的私有页面加入空闲链表。同样，当对页文件支持的区域最后一次访问结束时，这些页面也加入空闲链表。

当修改链表太大，或者零初始化和后备链表的大小低于一个最小值（例如在系统启动时计算出的内核变量MmMinimumFreePages所指定）时，被称为“修改页面写回器”的系统线程会被唤醒，将页面写回外存，并把这些页面移入后备链表。

2. 修改页面写回器

在修改页链表太大时，修改页面写回器通过将页面写回外存来限制链表的规模。它由两个系统线程组成：MiModifiedPageWriter将修改页写到页文件中；MiMappedPageWriter将修改页写入

映射文件。需要两个线程的目的是避免死锁。在没有空闲页可用时，如果映射文件的页面写入导致了一次缺页错误，并因此请求空闲页（这又需要修改页面写回器创建更多的空闲页），死锁就会发生。让修改页面写回器从另一个系统线程执行映射文件的页面调度I/O，这样线程就可以在不阻塞正常页文件I/O的情况下等待。

两个线程都以优先级17运行，并且在初始化后，等待各自的事件对象来触发。修改页面写回器事件被触发的原因有两个：

- 当修改页面的数量超越了系统初始化时计算出的最大值（MmModifiedPage Maximum）时
- 当可利用页数量（MmAvailablePages）小于MmMinimumFreePages时

表4-19列出了触发修改页面写回器减小修改链表大小的页面数量，以及链表中将留下的页面数量。和其他的内存管理变量一样，这个值也是在系统启动时依据物理内存数量计算出来的。

表4-19 修改页面写回器数值

内存规模	修改页面阈值	保留的修改页面数
<12 MB	100	40
12 ~ 19 MB	150	80
19 ~ 33 MB	300	150
>33 MB(特例)	400	800

另一个事件（MiMappedPagesTooOldEvent）也可以触发修改页面写回器。该事件在预定的数秒（MmModifiedPageLifeInSeconds）后产生，表示映射页面（不是修改页面）将被写入外存。这个值缺省为300秒（5分钟）。（可以通过添加DWORD型注册值HKLM\SYSTEM\CurrentControlSet\Control\SessionManager\MemoryManagement\ModifiedPageLife来替换缺省值）。设置这个事件是为了减少在修改链表没有达到表4-19中阈值的情况下，由于系统崩溃或断电所引起的数据丢失。

修改页面写回器一旦被激活，便试图利用一次I/O请求将尽可能多的页面写入外存。通过检查修改链表中页面的页框号数据库单元中初始页表项域，定位那些在外存上邻接的页面来实现这个目标。一旦邻接页面链表建立，页面就从修改链表中移出，并发出一次I/O请求。在I/O请求顺利完成后，这些页面会放置在后备链表尾部。

当另一个线程访问正在被写入的页面时，在描述物理页面的页框号项中的访问计数和共享计数会增加，表示另一个进程正在使用这个页面。当I/O操作完成时，修改页面写回器如判断共享计数不为0，就不会将此页移入后备链表。

3. 页框号数据结构

页框号数据库项是定长的，根据页面的状态而有几种不同的状态。因此，个别域针对不同的状态有不同的含义。页框号项的各种状态如图4-28所示。

对某几个页框号类型来说，有一些域是相同的，但其他域为特定页框号类型所特有。下列域出现在多种页框号类型中：

- **页表项地址** 指向此页的页表项的虚拟地址。



图4-28 页框号数据库项状态

- **访问计数** 对此页的访问数量。当页面首次被加入一个工作集且（或）当这页由于I/O在内存中锁定（例如，被一个设备驱动程序）时，访问计数就会增加。当共享计数为0或从内存中解锁时访问计数减少。访问计数为0时，该页不再属于工作集。于是，可以根据访问计数，更新描述该页面的页框号数据库项，以便将该页添加到空闲、后备、或修改链表中。
- **类型** 该页框号所表示的页面类型（活动/有效、过渡、后备、修改、修改不写入、空闲、零初始化和坏）。
- **标识** 标识域所包含的信息见表4-20。

表4-20 页框号数据库项中的标识

标 识	含 义
修改状态	表示此页是否被修改过。（如果此页被修改过，则它从内存移出前必须将其内容存入磁盘）
原型页表项	表示页框号项引用的页表项是原型页表项。（例如，此页是可共享的）
奇偶校验错误	表示该物理页包含奇偶校验或错误校正控制的错误
正在进行读取	表示对此页的页面调入操作正在进行。页框号中的第一个DWORD型数据包含了I/O完成时将被通知的事件对象的地址；也用于表示为非分页缓冲池分配的第一个页框号
正在进行写入	表示对此页的写操作正在进行。页框号中的第一个DWORD型数据包含了I/O完成时将被通知的事件对象的地址；也用于表示为非分页缓冲池分配的最后一个页框号

(续)

标 识	含 义
非分页缓冲池开始	对非分页缓冲池的页面，表示这是为非分页缓冲池分配的第一个页框号
非分页缓冲池终止	对非分页缓冲池的页面，表示这是为非分页缓冲池分配的最后一个页框号
页面调入错误	表示对此页进行页面调入时发生了一个I/O错误。(如果这样，页框号的第一个域会包含错误代码)

• **初始页表项的内容** 所有页框号数据库项均包括指向该页面的页表项(可能是一个原型页表项)的初始内容。保存页表项的初始内容是为了当物理页面不再驻留内存时，恢复该页表项。

• **页表项的页框号** 包含指向该页面页表项的页表页的物理页号。

剩余各域对各个类型的页框号来说是不同的。例如：图4-28中的第一种页框号表示活动的并且是某个工作集一部分的页面。共享计数表示指向该页的页表项数量。(标记为只读、写时复制或者是共享读/写的页面可以被多个进程共享。)对于页表页，这个域是页表中有效页表项的数量。只要共享计数大于0，该页就不能从内存中移出。

工作集索引域是一个进入进程(或系统)工作集链表的索引。如果是一个私有页面，工作集索引域直接指向工作集链表中的项，因为该页仅被映射到一个单独的虚拟地址上。如果是一个共享页面，工作集索引就是一个提示，它仅保证对于使页面有效的第一个进程是正确的。

图4-28中的第二种页框号对应于后备链表或者修改链表中的页面。在这里，前向和后向链接域将链表中的单元链接在一起。该链接很容易找到满足缺页错误的页面。当页面在上述两个链表之一时，共享计数确定为0(因为没有工作集使用该页)。然而，访问计数可能并不为0，因为进程中可能有对应于该页的I/O(例如，该页正在被写入外存)。

图4-28中的第三种页框号对应于空闲或零初始化链表中的页面。除了两个链表内的链接外，这些页框号数据库项还通过“颜色”(物理页面在处理器内存高速缓存中的位置)利用附加域来链接物理页面。Windows 2000/XP试图利用CPU高速缓存中不同的物理页面，来减少不必要的CPU存储器高速缓存的震荡。通过尽量避免让两个不同的页面避免使用相同的高速缓存项，来实现优化。对于带有直接映射高速缓存的系统，可以使性能得到显著提高。

图4-28中的第四种页框号表示正在进行I/O的页面(例如，页面读取)。当I/O正在进行时，页框号的第一个域指向在I/O完成时将被激活的事件对象。如果发生页面调入错误，这个域包含Windows 2000/XP表示I/O错误的错误状态代码。这个页框号类型用于解决冲突页错误。

除了页框号数据库，表4-21所示的系统变量还描述了物理内存的总体状态。

表4-21 表述物理内存系统变量

变 量	描述信息
MmNumberOfPhysicalPages	系统中可用的物理页面的总数
MmAvailablePages	系统中可用页面的总数——零初始化、空闲、和后备链表中页的总量
MmResidentAvailablePages	如果每个进程均处于它的最小工作集大小，是可利用的物理页面的总数

4.2.8 其他内存相关机制

1. 锁内存

通过两种方式可以将页面锁在内存中：

- 设备驱动程序可以调用核心态函数MmProbeAndLockPages, MmLock PagableCodeSection, MmLockPagableDataSection, 或者 MmLockPagableSection ByHandle。解锁之前，使用这种机制锁定的页面一直保持在内存中。
- Win32应用程序可以调用VirtualLock函数锁住进程工作集中的页面。注意，这种机制并不能防止调页——如果进程中所有的线程都处于等待状态，内存管理器就会根据需求自由地从工作集移出这些页面（对于修改页，将向磁盘写入该页）。这种情况下，在工作集中锁住页面实际上会降低性能，因为当一个线程被唤醒，并开始运行之前，内存管理器必须读入所有被锁的页面。因此，在大多数情况下，最好让内存管理器决定哪些页面留在物理内存中。一个进程可以锁住的页数量不能超过它的最小工作集大小减8。

2. 分配粒度

Windows 2000/XP按照系统分配粒度(allocation granularity)定义的整型边界，对齐每个保留的进程地址空间区域，系统分配粒度值通过Win32 函数GetSystemInfo找到。目前，这一值为64 KB。选定这个值是为了支持将来具有更大页面规模的处理器，对于那些假定分配对齐方式的应用程序来说，也减少了改变的风险。（Windows 2000/XP核心态代码不适用于同样的限制，它可以在页粒度上保留内存。）

保留地址空间区域时，Windows 2000/XP保证了区域大小是系统页大小倍数。例如，因为x86系统使用4 KB的页，如果你试图保留18 KB大小的内存区域，实际存储在x86系统上的数量将会是20 KB。

3. 内存保护机制

Windows 2000/XP提供了内存保护机制，防止用户进程无意或有意地破坏另一进程或操作系统的地址空间。Windows 2000/XP通过4种基本的方式来提供保护。

首先，所有系统范围内核心态组件使用的数据结构和内存缓冲池只能在核心态下访问，用户态线程不能访问这些页面。如果它们试图这样做，硬件会产生一个错误信息，随后内存管理器向线程报告一个访问冲突。

第二，每个进程有一个独立、私有的地址空间，禁止其他进程的线程访问。唯一例外是，该进程和其他进程共享页面，或另一进程具有对进程对象的虚拟内存读写权限，可以使用ReadProcessMemory或WriteProcessMemory函数。当线程访问一个地址时，通过控制虚拟地址的转换，Windows 2000/XP可以保证在进程中运行的线程不会错误地访问属于另一进程的页面。

第三，除了提供虚拟到物理地址转换的隐含保护外，所有Windows 2000/XP支持的处理器还提供了一些硬件内存保护措施（如读/写，只读等）。这种保护的细节根据处理器不同而不同。例如，在进程的地址空间中代码页被标志为只读，可以防止被用户线程修改。

最后，共享内存区域对象具有标准的Windows 2000/XP存取控制表（ACL），当进程试图打

开它们时会检查ACL表，这样对共享内存的访问也限制在具有适当权限的进程之中。

4. 写时复制

写时复制页面保护是内存管理器用来节约物理内存的优化技术。当进程映射区域对象的写时复制视图时，内存管理器延迟到页面修改时才进行复制，而不是在映射视图的同时制作进程私有副本（像Compaq Open VMS操作系统那样）。现代UNIX系统也使用了这一技术。例如，图4-29所示，两个进程共享3个页，每个页都被标志成写入时复制，但是没有任何一个进程试图改变页上的数据。

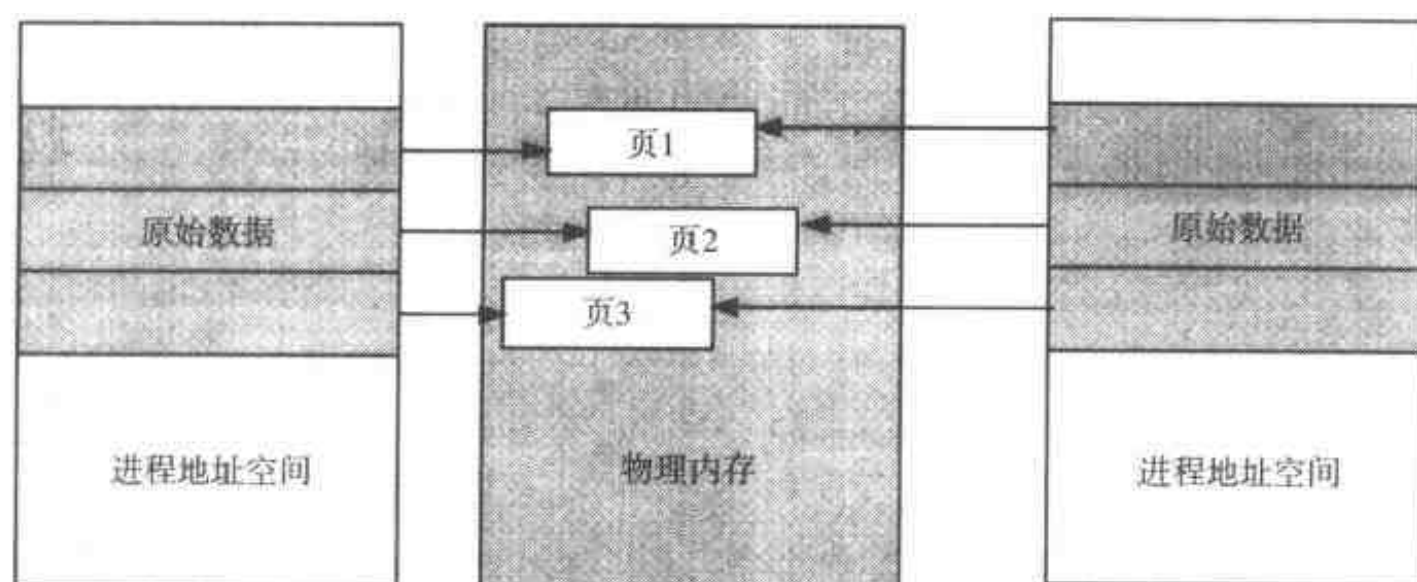


图4-29 写时复制之前

如果任何一个进程中的线程写入这些页面，就会产生内存管理错误。内存管理器观察到写操作是作用于写时复制页面，因此它不以访问冲突来报告这个错误，而是在物理内存中分配一个新的读/写页，并将原始页面的内容复制到新页上，更新该进程相应的页面映射信息以指向新位置，并消除此错误，使产生错误的指令重新执行。此时就可以成功地进行写操作，但如4-30图所示，新复制的页面对于做写操作的进程来说是私有的，对于其他共享写时复制页面的进程是不可见的。

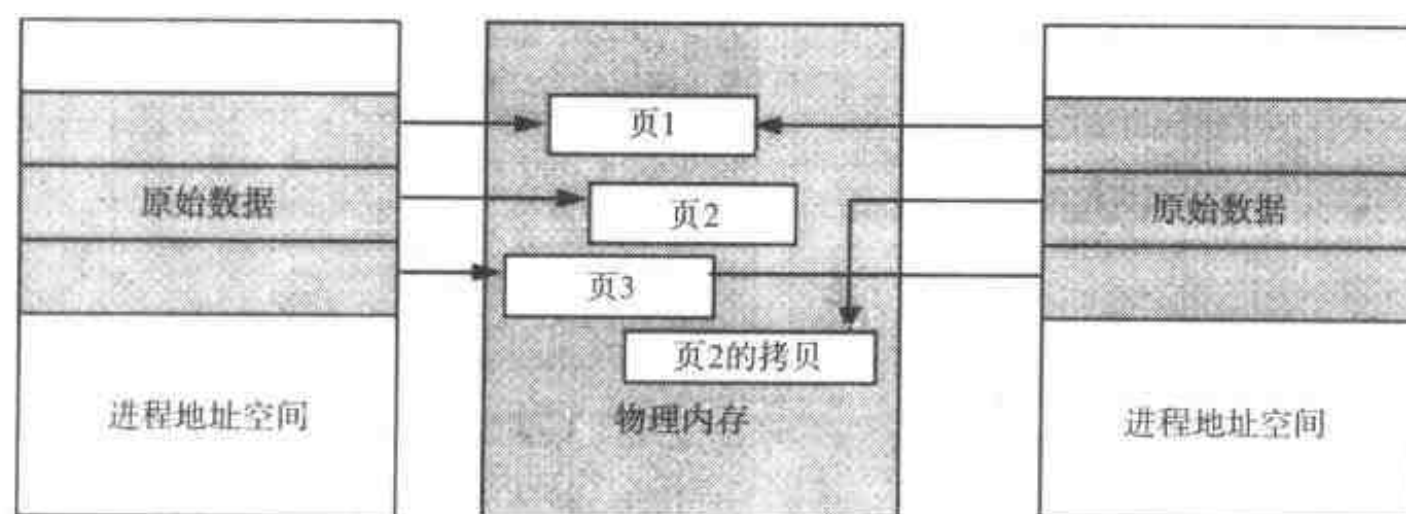


图4-30 写时复制之后

5. 地址窗口扩充

虽然Windows 2000/XP系统能够支持64 GB物理内存，但每个32位的用户进程只有2 GB或3 GB虚拟地址空间（依赖于3 GB开关是否被激活）。为了使32位进程能够访问更多的物理内存，

Windows 2000/XP提供了一个叫做地址窗口扩充(address windowing extension, AWE)的函数集。例如, 在一个有8 GB物理内存的高级Windows 2000/XP服务器上, 数据库服务器应用程序可以使用AWE来分配和使用将近8 GB的内存作为数据库缓存。

可以按以下三步使用AWE函数来分配内存:

- 1) 分配将要使用的物理内存。
- 2) 创建一个虚拟地址空间区域作为窗口来映射物理内存视图。
- 3) 将物理内存视图映射到窗口。

分配物理内存时, 应用程序可以调用Win32函数AllocateUserPhysicalPages (调用这函数需要锁内存页面的用户权限。)。应用程序然后使用Win32的VirtualAlloc函数和MEM_PHYSICAL标志在进程私有地址空间中创建一个窗口, 用来映射事先分配好的物理内存。AWE分配的内存几乎可以被所有Win32 API使用。(但是微软的DirectX函数不能使用。)

如果应用程序在它的地址空间里创建了一个256 MB的窗口, 并分配了4 GB的物理内存(在多于4 GB物理内存的系统上), 那么这个程序可以使用MapUserPhysicalPages或者MapUserPhysicalPagesScatter Win32函数, 通过将内存映射到256 MB窗口来访问物理内存。在给定映射下, 应用程序虚拟地址空间窗口的大小决定了应用程序可以访问的物理内存数量。图4-31显示了一个在服务器应用程序地址空间的AWE窗口, 该窗口映射到由AllocateUserPhysicalPages分配的物理内存。

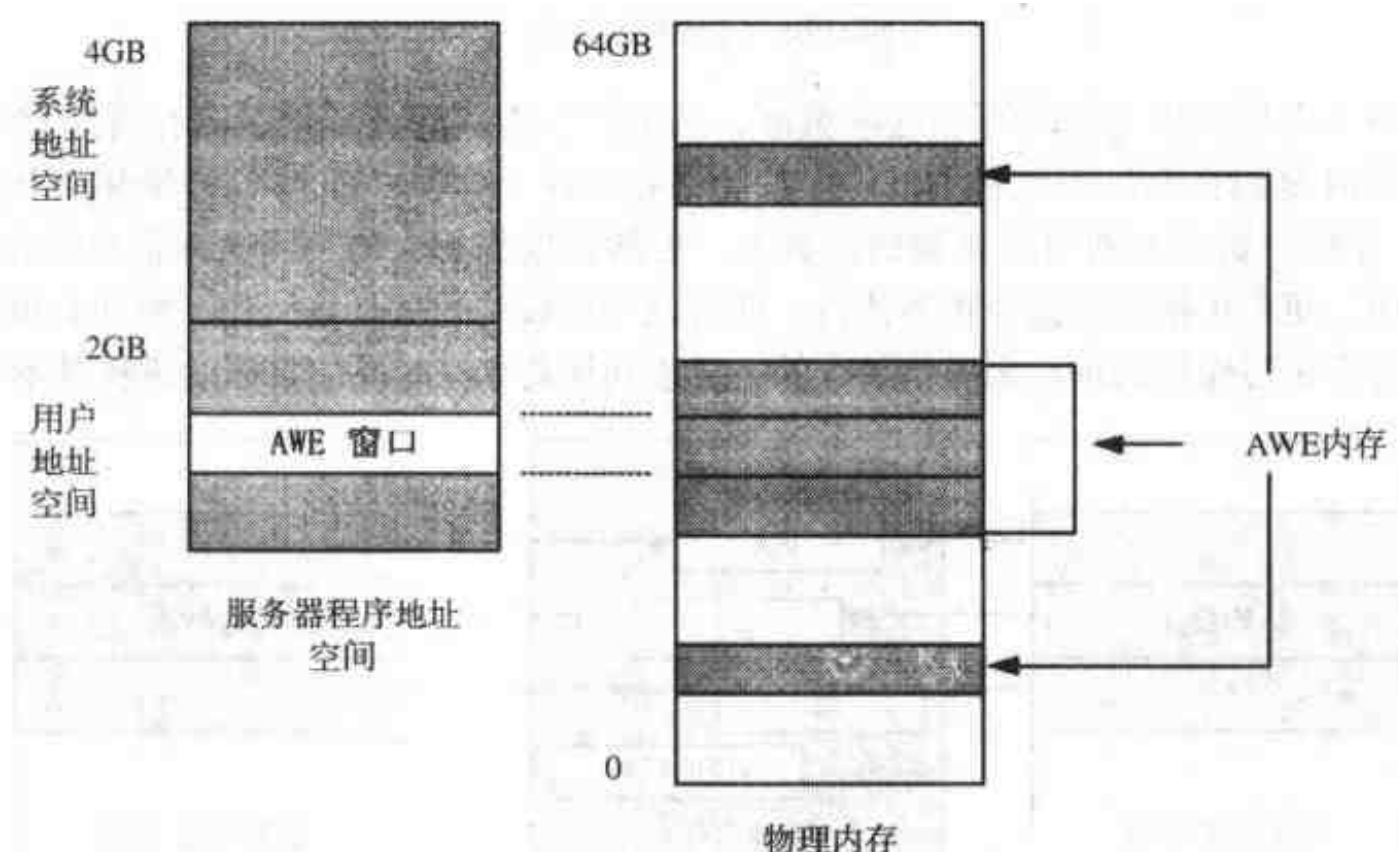


图4-31 使用AWE映射物理内存

所有Windows 2000/XP版本都包含AWE函数, 不管系统有多少物理内存这些函数都是可用的。然而, AWE对于多于2 GB物理内存的系统是最有用, 因为对于32位进程来说, 这是唯一的可以直接使用多于2 GB内存的方式。

下面是有关AWE函数分配和映射内存的限制。

- 页面不能在进程间共享。
- 同一物理页面不能映射到同一进程的多个虚拟地址上。
- 页保护是可读/写。

下面将介绍映射4 GB以上物理内存的页表数据结构。

6. 物理地址扩展

Intel x86系列的处理器，自Pentium Pro之后都包含一个称为“物理地址扩展”（physical address extension, PAE）的内存映射模式。物理地址扩展模式允许访问64 GB的物理内存。一旦x86处理器运行在物理地址扩展模式下，内存管理单元（memory management unit, MMU）就将虚拟地址划分为4个域，如图4-32所示。

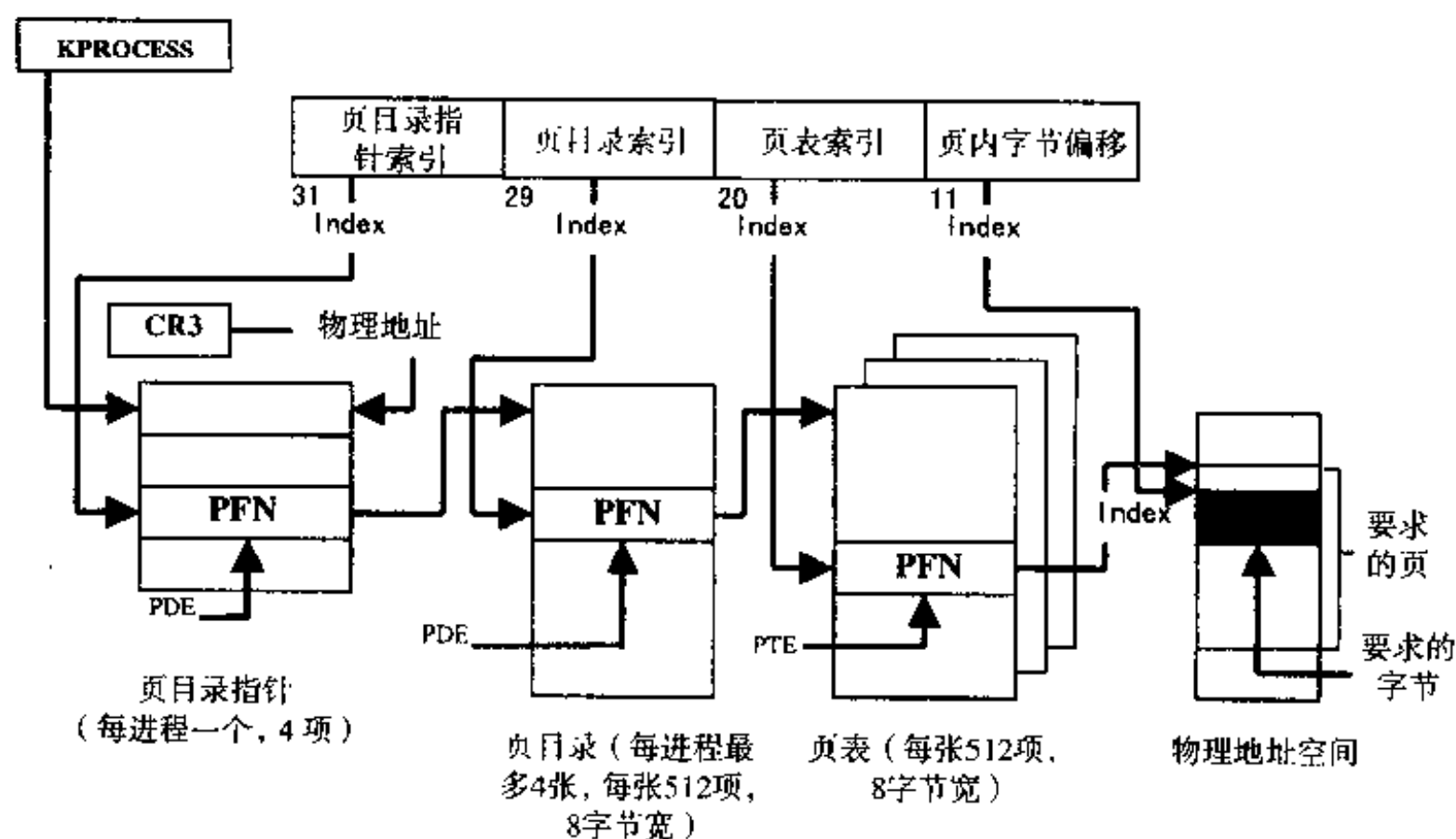


图4-32 PAE模式的页映射

内存管理单元除了使用页目录和页表外，还有一个更高层次的表——页目录指针表。与标准的变换模式相比，物理地址扩展模式能对更大范围的内存进行编址，这不是因为多了一个额外的转换层次，而是因为页目录项和页表项都由32位扩展到了64位。x86处理器通过设置CR4寄存器中的PAE标志，将另外4根地址线并入当前32位物理地址，这就使得x86处理器有能力支持2³⁶字节，或者64 GB大小的内存。

核心内核映像（Ntoskrnl.exe）有一个支持PAE的特别版本叫作Ntkrnlpa.exe。（多处理器版本叫作Ntkrnpamp.exe。）为了使用支持PAE的内核，必须在Boot.ini中加入/PAE选项。

4.3 Windows 2000/XP外存管理

外存管理定义了操作系统与非易失性的存储设备和介质的接口方式。外存包括许多不同的设备，如磁带设备，光介质，CD唱机，软盘，硬盘。Windows 2000/XP对上述每一种存储介质都

提供支持。因为我们这本书主要研究Windows 2000/XP的内核组件，所以这部分重点研究Windows 2000/XP硬盘存储子系统。

我们首先定义基本盘(basic disk)和动态盘(dynamic disk)，并解释它们是如何分区的。其次介绍了核心态的设备驱动程序是如何与文件系统的驱动程序接口操纵硬盘设备的。接着介绍Windows 2000/XP多分区磁盘管理的特性是如何实现的，包括在不同的物理盘上为提高可靠性和增强性能而复制和分割文件系统数据。最后讨论Windows 2000/XP分配驱动器名(drive letter)的过程以及文件系统的驱动程序如何安装(mount)卷(volume)。

4.3.1 Windows 2000/XP存储的演变

Windows 2000/XP存储系统是从微软的第一个操作系统MS-DOS演变而来。由于硬盘变得越来越大MS-DOS也要适应这种趋势。微软改进的第一步就是，让MS-DOS在一个物理盘上采用多个分区，也就是逻辑盘。MS-DOS可以把每个分区格式化为不同的文件系统类型(FAT12或FAT16)，而且为每个分区分配一个不同的驱动器名。MS-DOS版本3和4严格限制分区的数量和分区的大小，但是到了MS-DOS 5这种分区机制就完全成熟了。MS-DOS 5可以把硬盘分为任意多的区，每个区可以任意大小。

Windows NT借鉴了MS-DOS的分区机制。这一方面是为了提供一个与MS-DOS和Windows 3.X相兼容的磁盘，另一方面是为了让开发小组可以使用一些已有的磁盘管理工具。微软在Windows NT中扩展了MS-DOS分区的基本概念，支持企业级操作系统所需的一些存储管理的特征：跨磁盘管理(disk spanning)和容错(fault tolerance)。从Windows NT的第一个版本3.1开始，系统管理员就可以创建由多个分区组成的卷，这样就允许一个大的卷由分布在不同物理盘上的分区组成，从而利用以软件为基础的数据冗余实现容错。

与早期Windows NT所支持的MS-DOS的分区方式相比，虽然这种方式已经可以胜任多数的存储管理任务，但它也有几个缺点。一个缺点是对大多数磁盘设置的改变需要重启操作系统才能生效。对今天的服务器来说，它们必须持续服务若干个月甚至几年。任何一次重新启动，甚至包括计划好的重新启动，都是极不方便的。另一个缺点是Windows NT的注册表中为MS-DOS方式的分区保存了多分区磁盘的配置信息，这种方式导致在不同的系统间移动磁盘时，移动配置信息变成一项非常艰巨的任务。当你重新安装操作系统时，也很容易丢失配置信息。最后，每个卷有一个唯一的从A到Z的驱动器名，也让使用Windows 2000/XP以前的操作系统的用户，受到这样的困扰：用户可创建的本地或远程的卷数目有一个上限。

为了完全理解本章的剩余部分，需要了解如下的基本术语。

- 盘(disk)是一种物理存储设备，如硬盘，3.5英寸软盘，光盘。
- 盘被分为扇区(sector)，这是可寻址的大小固定的块。
- 分区是盘上连续扇区的集合，分区表或其他盘管理数据库中保存了分区的起始扇区、大小和其他特性。
- 简单卷(simple volume)是这样一种对象，它代表文件系统驱动程序作为一个独立单元管理来自一个分区的所有扇区。

- 多分区卷 (multipartition volume) 是这样一种对象, 它代表文件系统驱动程序作为一个独立单元管理来自多个分区的所有扇区。多分区卷提供简单卷所不支持的性能、可靠性和大小等特性。

4.3.2 分区

Windows 2000/XP 引入了基本盘和动态盘的概念。Windows 2000/XP 把基于MS-DOS分区方式的盘称为基本盘。从某种意义上说, 基本盘是Windows 2000/XP继承的盘。动态盘对Windows 2000/XP来说是一个新的概念, 它实现了比基本盘更具适应性的分区机制。基本盘和动态盘之间一个主要的不同点在于, 动态盘支持创建新的多分区卷。(基本盘只支持从Windows NT4升级的多分区卷) 回忆一下上一节中的那些术语, 多分区卷提供简单卷所不支持的性能、可靠性和大小等特性。基本盘的多分区卷配置信息保存在注册表中。动态盘的多分区卷配置信息保存在磁盘中。将多分区卷的配置信息保存在磁盘中而不是在注册表中, 可以把动态盘和它所描述的存储介质联系在一起, 这样就不容易丢失数据, 并且在系统间移动磁盘也变得更简单。

Windows 2000/XP把所有盘当做基本盘来管理, 除非手工创建一个动态盘, 或者把已经存在的基本盘(其中要有足够的空间)转变成动态盘。为了鼓励系统管理员应用动态盘, 微软给基本盘加了一些限制。例如, 你只可以在动态盘上创建一个新的多分区卷。另一个限制是Windows 2000/XP只在动态盘上实现NTFS的动态扩容。动态盘的一个缺点是, 它所采用的分区格式是专有的, 并且不和其他的操作系统兼容, 包括其他版本的Windows。所以你不能在一个双引导的环境中访问动态盘。

1. 基本分区

在一台计算机上安装Windows 2000/XP时, 必须先做的一件事就是在系统的主物理盘上创建一个分区。Windows 2000/XP在其上定义了系统卷, 用于存储引导过程中用到的文件。另外, Windows 2000/XP的安装过程也需要创建一个分区用来当做引导卷, Windows 2000/XP在引导卷上安装系统文件和创建系统目录(\Winnt)。系统卷和引导卷可以是同一个卷, 这样不必为引导卷创建一个新的分区。

x86硬件系统采用的BIOS标准规定了Windows 2000/XP分区格式必须遵守的一个要求, 即主盘的第一个扇区中包含主引导记录(MBR)。当x86处理器开始引导的时候, 计算机的BIOS读取主引导记录中的内容, 并把它当做可执行代码。BIOS完成硬件的基本设置后, 激活MBR代码启动操作系统的引导过程。在微软的操作系统中, 包括Windows 2000/XP, 它们的主引导记录中包含了一个分区表。一个分区表有四个项, 它最多可定义四个主分区(primary partition)的位置。分区表也记录了分区的类型。有很多预先定义的分区类型存在, 分区类型指定了分区中包含的文件系统(例如, FAT32和NTFS)。一个特别的分区类型是扩展分区(extended partition), 它包含另一个主引导记录, 内有其自身的分区表。由于采用了扩展分区, 使微软的操作系统克服了一个磁盘只能有四个分区的限制。一般来说扩展分区可以反复无限地使用, 这意味着一个磁盘上可能存在的分区数是没有限制的。Windows 2000/XP引导过程明确地区分主分区和扩展分区。系统必须将主盘上的一个主分区标记为活动, 这样Windows 2000/XP在主引导记录中的代码装入活动分区

第一个扇区中的代码，并把控制交给它。由于主分区的第一个扇区在引导过程中起了重要作用，Windows 2000/XP指定任何一个分区的第一个扇区为引导扇区。每个被格式化为某种文件系统的分区都有一个引导扇区用来存储这个分区上文件系统结构的信息。

2. 动态分区

动态盘是Windows 2000/XP偏爱的磁盘格式，也是创建新的多分区卷所必需的。Windows 2000/XP的逻辑磁盘管理子系统(LDM)负责管理动态盘。LDM的分区与MS-DOS分区的一个主要的不同点在于，LDM维护一个单独的数据库用来储存系统动态盘的分区信息，包括多分区卷的设置。

LDM的数据库存在于每个动态盘最后的1 MB保留空间中。正是因为需要这个空间，Windows 2000/XP在将一个基本盘转化为动态盘时，需要在基本盘的最后有一定的剩余空间。LDM也实现了一个MS-DOS的分区表，这是为了继承一些在Windows 2000/XP下运行的磁盘管理工具，或是在双引导环境中让其他系统不至于认为动态盘还没有被分区。由于LDM分区在磁盘的MS-DOS分区表中并没有体现出来，所以被称为软分区，而MS-DOS分区被称为硬分区。

另一个LDM创建MS-DOS分区表的原因是为了Windows 2000/XP引导程序能够找到系统卷和引导卷，即使它们在动态盘上（例如，Ntldr就不知道LDM分区的存在）。如果一个盘中包括系统卷和引导卷，MS-DOS分区表中的硬分区将描述这些卷的位置。否则，硬分区在磁盘的第一个柱面开始一直延伸到LDM的数据库。LDM指定这个分区为LDM类型——Windows 2000/XP新引进的MS-DOS风格分区类型。保存在MS-DOS风格分区中的区域就是LDM创建软分区的地方。图4-33用来说明动态盘。

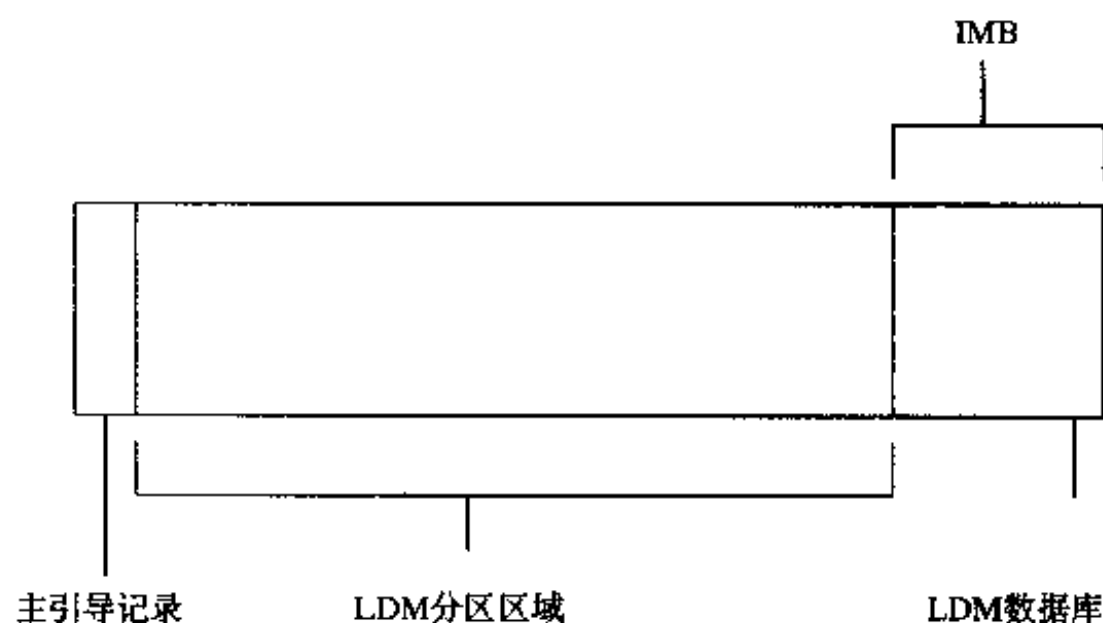


图4-33 动态盘的内部组织

LDM的数据库中包含四个区域，图4-34说明了这一点，一个头扇区（LDM称它为私有头），一个内容表的区域，一个数据库记录区和一个事务处理日志区（图4-34中所示的第五个区是私有头的一个简单的复制）。私有头扇区存在于动态盘最后1MB的位置上，是数据库的标志。Windows 2000/XP用GUID区分所有的东西，磁盘也不例外。一个GUID是一个128位的值用于区分Windows 2000/XP中不同的对象。LDM给每一个动态盘分配一个GUID，私有头扇区记录了这个GUID。

私有头中也存放了磁盘组的名字（该名字是由Dg0和计算机的名字一起组成，例如SusanDg0，意味着计算机的名字是Susan）和一个指向数据库内容表的指针。为了保证可靠性，LDM在磁盘的最后一个扇区保存了一个私有头的拷贝。

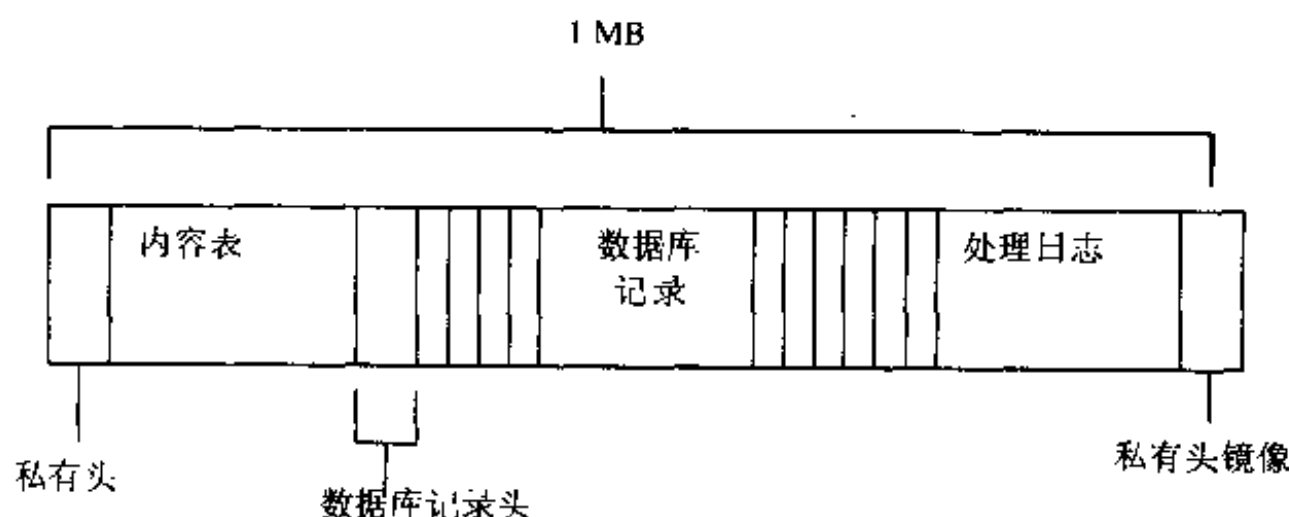


图4-34 LDM的数据库

数据库内容表有16个扇区大小，其中包含关于数据库布局的信息。LDM让数据库记录区域紧接着内容表，并将内容表后第一个扇区作为数据库记录头。这个扇区中存储了数据库记录区的的信息，包括其所包含的记录个数，数据库相关的磁盘组的名字和GUID，以及LDM用于创建下一项的序列号。数据库记录头后的扇区中是128位定长记录，用来描述磁盘组的分区和卷。

数据库中的每一项可以是如下四种类型之一：分区，磁盘，组件，卷。LDM把每一项与内部对象的标识符联系到一起。在最低的级别，分区项描述软分区，它是在一个盘上的连续区域。存储在分区项中的标识符把这个项与一个组件和一个磁盘项联系起来。磁盘项代表一个磁盘组中的动态盘，包括磁盘的GUID。组件项像一条链子把一个或多个分区项和与分区相连的卷项联系起来。卷项存放这个卷的GUID，卷的大小和状态，驱动器的名字。比一个数据库记录大的磁盘项占用多个记录的空间，分区项、组件项和卷项很少占用多个记录的空间。

LDM需要三个项来描述一个简单卷：分区项、组件项和卷项。分区项描述系统分配给某个卷的磁盘上的一个区域；组件项把一个分区项和一个卷项联系起来；卷项中包含Windows 2000/XP内部用来识别卷的GUID。多分区卷需要的项数多于三个。例如，一个条带卷（条带卷后面将要讲到）包括最少两个分区项、一个组件项和一个卷项。唯一一种含有一个以上组件项的卷的类型是：镜像卷。镜像卷含有两个组件项，每个只表示这个镜像的一半。LDM为每个镜像卷使用两个组件项的目的是：当一个镜像破坏时LDM能够在组件一级将他们分割开来，并创建两个各含有一个组件项的卷。因为简单卷需要三个项，而1 MB数据库空间大约可以容纳8000个项，所以在Windows 2000/XP中可以创建的卷数目的有效上界大约是2500个。

LDM数据库的最后部分是事务处理日志区，它包含的几个扇区在数据库信息改变时用来存储备份信息。这样确保在系统崩溃或断电时，LDM能够利用日志把系统恢复到一个正确的状态。

4.3.3 驱动程序

Ntldr是Windows引导最初所使用的操作系统文件。系统卷中引导扇区中的代码负责执行

Ntldr。Ntldr从系统卷中读取Boot.ini文件，把计算机的引导选项显示给用户。Boot.ini指定分区名为mult(0)disk(0)rdisk(0)partition(1)的形式。这些名字是按ARC (Advanced RISC Computing) 规范命名，它们是Alpha的固件和其他RISC处理器使用的分区命名标准。Ntldr把Boot.ini中用户指定的项转换为正确的引导分区，然后将Windows 2000/XP系统文件（从注册表、Ntoskrnl.exe、引导驱动程序开始）装入内存，继续引导过程。

1. 磁盘驱动程序

在初始化过程中，Windows 2000/XP I/O管理器启动硬盘的存储驱动程序。Windows 2000/XP的存储驱动程序，遵循类、端口、小端口 (class/port/miniport) 的结构。微软提供一个类驱动程序实现所有存储设备共同的功能；提供一个端口的驱动程序实现基于某种特定总线设备的共同功能，如SCSI、IDE；OEM提供小端口驱动程序，加入端口驱动程序。

在磁盘存储驱动程序体系结构中，只有类驱动程序必须遵循Windows 2000/XP设备驱动程序的接口，小端口驱动程序使用端口驱动程序接口，而不是使用设备驱动程序接口。端口驱动程序只是实现了一些设备驱动程序的支持例程，这些例程作为小端口驱动程序与Windows 2000/XP之间的接口。微软提供操作系统专用的端口驱动程序，使小端口驱动程序在Windows 98，Windows ME，Windows 2000/XP 之间二进制兼容，因此使小端口驱动程序开发者的工作得到简化。Windows 2000/XP磁盘类(\Winnt\System32\Drivers\Disk.sys)，是实现所有磁盘功能的类驱动程序。Windows 2000/XP也提供了一些磁盘的端口驱动程序。

2. 设备命名

Windows 2000/XP磁盘类驱动程序创建代表磁盘和分区的设备对象。代表磁盘的设备对象有\Device\HarddiskX\DRX这样的名字，X代表磁盘标号。磁盘的类驱动程序使用I/O管理器的IoReadPartitionTable函数识别表示分区的设备对象。由于小端口设备驱动程序为磁盘类驱动程序提供在引导过程识别出的磁盘，磁盘类驱动程序可以为每个磁盘调用IoReadPartitionTable函数，IoReadPartitionTable函数调用类、端口、小端口驱动程序提供的扇区一级的磁盘I/O，读取MS-DOS格式的磁盘分区和建立磁盘硬分区的内部表示。磁盘类驱动程序为每个从IoReadPartitionTable函数得到的主分区创建设备对象（包括扩展分区中的主分区）。下面是一个分区对象名字：

\Device\Harddisk0\DP(1)0x7e000-0x7ff50c00+2

这个名字表示系统的第一个盘上的第一个分区。前两个十六进制数字 (0x7e000-0x7ff50c00) 指定了分区的起始位置和长度，最后一个数字是类驱动程序分配的内部标识符。

为了与使用Windows NT4命名习惯的应用程序保持兼容，磁盘的类驱动程序创建Windows NT4格式的符号连接，用来引用驱动程序创建的设备对象。例如，磁盘的类驱动程序创建连接\Device\Harddisk0\Partition0，用来引用\Device\Harddisk0\DR0；创建连接\Device\Harddisk0\Partition1，用来引用第一个磁盘的第一个分区的设备对象。类驱动程序也在Windows 2000/XP中创建同样的Win32符号连接，代表在Windows NT4系统下创建的物理驱动器。例如，创建连接\??\PhysicalDrive0，用来引用\Device\Harddisk0\DR0。

Win32 API并不知道Windows 2000/XP对象管理器的名字空间，Windows 2000/XP保存了两个不同的名字空间子目录供Win32使用，其中之一是\??子目录（另一个是\BaseNamedObjects子

目录)。在\??子目录中, Windows 2000/XP创建了一些与Win32程序交互的硬件对象, 包括串口和并口, 还有磁盘。由于磁盘对象实际上存在于其他的子目录中, 所以Windows 2000/XP使用符号链接, 把在\??子目录下的名字与在名字空间其他地方的对象联系起来。I/O管理器为系统中的每一个物理盘都创建一个\??\PhysicalDriveX的链接, 指向\Device\HarddiskX\Partition0 (从零开始的数字来替代X)。那些直接访问磁盘扇区的Win32应用程序可以调用Win32函数CreateFile, 通过指定\\.\PhysicalDriveX (X是一个磁盘的号码) 作为参数来打开磁盘。Win32 的应用层先把名字转化为\??\PhysicalDriveX, 然后在把名字提交给Windows 2000/XP对象管理器。

3. 基本盘的管理

FtDisk驱动程序 (\Winnt\System32\Drivers\Ftdisk.SYS) 创建代表基本盘上卷的磁盘设备对象。在WINNT4中, 至少在有一个多分区卷存在时, FtDisk才存在。在Windows 2000/XP中, FtDisk在对所有基本盘的卷 (包括简单卷) 的管理都起着不可缺少的作用。FtDisk为每一个卷都创建一个形如\Device\HarddiskVolumeX的设备对象, 其中X是一个从1开始的数字, 用来标识这个卷。FtDisk用存储在注册表HKLM\SYSTEM\Disk中的基本盘配置信息, 来决定基本卷、多分区卷和系统卷所包含的内容。注意: 磁盘键只存在于从Windows NT4和从Windows 98升级而来的Windows 2000/XP系统中, 原系统磁盘管理需要有一个磁盘键 (如分配卷标或创建多分区卷)。为了避免对注册表的依赖, FtDisk把在磁盘键中的配置信息, 移到了磁盘上的隐藏扇区中。

FtDisk实际上是一个总线驱动程序, 在分区管理驱动程序 (Partmgr.sys) 的帮助之下列出基本盘, 找出已存在的基本卷并把信息报告给Windows 2000/XP的即插即用管理器。分区管理器向即插即用管理器进行注册, 因此Windows 2000/XP在磁盘类驱动程序创建分区设备对象时能够通知分区管理器。分区管理器通过一个私有的接口将这个新分区对象通知给FtDisk, 并且创建一个与分区对象捆绑在一起的过滤设备对象。过滤设备对象的存在可以提示Windows 2000/XP当一个分区设备对象被删除时通知分区管理器, 以便分区管理器能够更新FtDisk。在微软管理控制台 (MMC, Microsoft Management Console) 磁盘管理工具中磁盘分区被删除时, 磁盘类驱动程序删除这个分区设备对象。由于FtDisk了解分区信息, 它利用基本盘的配置信息来确定分区和卷的一致性, 并且当知道一个卷所描述的所有分区都存在时, 它便创建一个卷设备对象。

前面简短讨论过的Windows 2000/XP驱动器名分配过程, 在对象管理器的目录\??下创建一个驱动器名的符号连接指向FtDisk创建的卷设备对象。当系统或者应用程序第一次访问一个卷时, Windows 2000/XP将执行安装操作, 文件系统可以识别和声明对格式化为某种类型的卷的所有权 (安装操作在本章后面卷的安装一节中描述)。

4. 动态盘管理

MMC磁盘管理工具的动态连接库 (DMDiskManager, 位于Winnt\System32\Dmdiskmgr.dll) 可以使用DMAdmin.EXE、LDM的磁盘管理服务 (Winnt\System32\Dmadmin.exe) 来创建和更改LDM数据库的内容。运行开始使用MMC磁盘管理工具时, DMDiskManager装入内存, 启动DMAdmin。DMAdmin从磁盘上读取LDM的数据库信息, 并把它所取得的信息返回给

DMDiskManager。如果DMAdmin发现另一个计算机磁盘组的数据库，则将这个盘上的卷标记为外部，这样当用户想使用时，可以把这个盘加入到当前计算机的数据库。改变动态盘的设置时，DMDiskManager会通知DMAdmin，DMAdmin改变数据库在内存中的拷贝。当DMAdmin完成这些改变时，它把更新的数据库传给DMIO（Dmio.sys设备驱动程序）。DMIO是FtDisk在动态盘的对应，所以它控制对磁盘上数据库的访问和创建代表动态盘上卷的设备对象。当你退出磁盘管理时，DMDiskManager停止和卸载DMAdmin服务。

DMIO并不知道如何解释它所监视的数据库。DMConfig（Winnt\System32\Dmconfig.dll，DMAdmin装载的一个DLL），和另一个设备驱动程序DMBoot（Dmboot.sys），负责解释这个数据库的内容。DMConfig知道如何读取和更新数据库，DMBoot只知道如何读取这个数据库。如果另一个LDM驱动程序DMLoad（Dmload.SYS）发现系统中至少有一个动态盘存在，DMBoot在引导中就被装入内存。DMLoad通过询问DMIO来做出这个决定：如果至少一个动态盘存在，DMLoad启动DMBoot，用它来扫描LDM数据库。DMBoot通知DMIO它所遇到的每个卷的组成，DMIO可以创建用来表示每个卷的设备对象。当DMBoot完成它扫描的任务后立即从内存中卸载。由于DMIO中没有解释数据库的部分，它相对较小，因此可以长驻内存。

和FtDisk一样，DMIO是一个总线驱动程序，它为动态盘的每一个卷创建一个名如\Device\HarddiskDmVolumes\PhysicalDmVolumes\BlockVolumeX的设备对象，其中X是DMIO分配给每一个卷的标识符。另外，DMIO创建另一个设备对象代表卷的初始（无结构）I/O，叫做\Device\HarddiskDmVolumes\PhysicalDmVolumes\RawVolumeX。DMIO也为每一个卷在对象管理器的名字空间内创建了许多符号连接。每个卷从形如\Device\HarddiskDmVolumes\ComputerNameDg0\VolumeY的连接开始。DMIO用计算机的名字来代替ComputerName项，用卷标识符来替代Y（与DMIO分配给设备对象的内部标识符不同）。这些连接用来引用PhysicalDmVolumes目录下的块设备对象。

4.3.4 多重分区管理

FtDisk和DMIO负责识别文件系统驱动程序管理的卷，并将I/O直接从卷映射到组成卷的底层分区。对简单卷来说，通过把卷的偏移量加上卷在磁盘中的起始地址，卷管理器可以保证卷的偏移量被转换成盘的偏移量。

对于多分区卷这就复杂多了，因为组成卷的分区可以是不邻接的分区，甚至可以在不同的磁盘中。有一些多分区卷使用数据冗余技术，所以它们需要更多的卷到磁盘的转换工作。所以，FtDisk和DMIO必须处理所有的多分区卷I/O请求，并确定这些I/O最终影响的分区。

在Windows 2000/XP中，有如下几种类型的多分区卷：

- 跨分区卷(spanned volume)
- 镜像卷(mirrored volume)
- 条带卷(striped volume)
- 廉价冗余磁盘阵列5卷(RAID-5 volume)

在描述每种类型多分区卷的分区配置和逻辑操作后，我们将研究FtDisk和DMIO对处理文件

系统发给多分区卷I/O请求包的方式。由于FtDisk和DMIO支持相同的多分区卷类型，在解释多分区卷的整个过程中“卷管理器”这个术语将用来指代它们。

1. 跨分区卷

跨分区卷是一个单独的逻辑卷，最多由在一个或多个磁盘上的32个空闲分区组成。Windows 2000/XP MMC磁盘管理工具把这些分区组成一个跨分区卷。这种卷可以被格式化为Windows 2000/XP所支持的各种分区类型。图4-35展示了驱动器号为D的100MB跨分区卷，它由第一个盘的后三分之一和第二个盘的前三分之一组成。跨分区卷在Windows NT 4中被称为卷集。

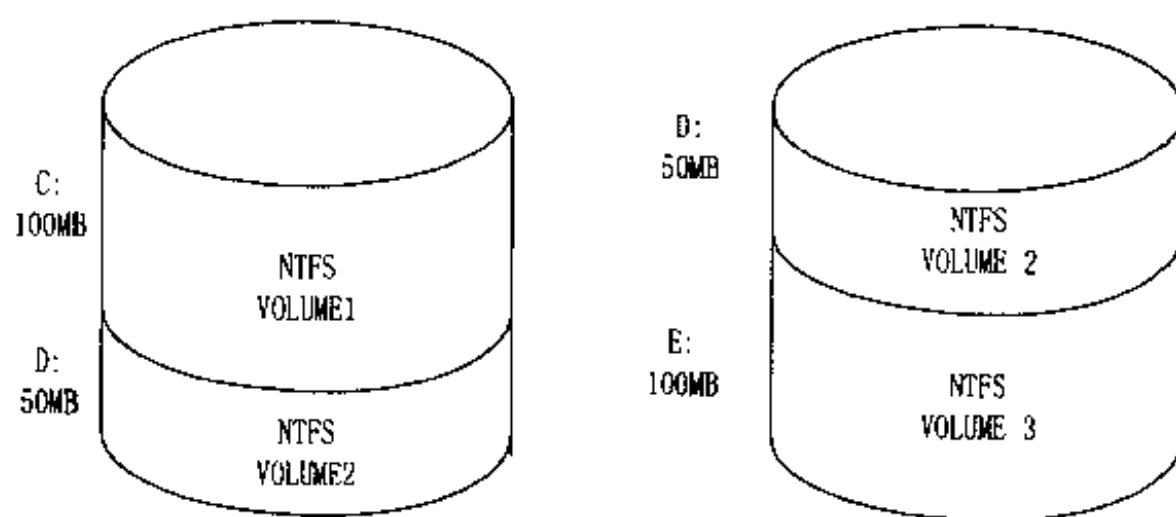


图4-35 跨分区卷

跨分区卷可以用来把小的磁盘空闲区域，或者把两个或更多的小磁盘组成大的卷。如果跨分区卷被格式化为NTFS，那么它在扩展空闲空间或者磁盘时，不影响已经被保存在其上的数据。这种可扩展性就是把在NTFS卷上的所有数据描述成一个文件带来的最大好处。NTFS能够动态增加逻辑卷容量的原因是用来记录卷分配状态的位图也是一个文件，即位图文件。通过扩展位图文件可以在卷中加入任意空间。另一方面，动态扩展FAT卷需要扩展FAT本身，这将导致丢失磁盘上所有信息。

卷管理器对Windows 2000/XP的文件系统隐藏了磁盘物理配置信息。例如，NTFS把图4-35中的卷D当做一个普通的100MB卷。NTFS通过查看它的位图文件来确定其中哪些空间是可被分配的。接着调用卷管理器来读写在卷中某偏移处的数据。卷管理器把跨分区卷中的物理扇区看成从第一个盘的第一个空闲区域到最后-一个盘的最后一个空闲区域是连续的，由它来确定偏移量所指扇区应该是哪个物理盘上的哪个扇区。

2. 条带卷

条带卷是一系列分区组成的单独的逻辑卷，最多有32个分区并且每个盘一个分区。条带卷也被称为RAID-0卷。图4-36展示了一个在三个盘上由三个分区组成的条带卷，每盘一个分区。（条带卷中的一个分区不需要占据整个磁盘，唯一的限制是每个盘上的分区大小相同。）

对文件系统来说，条带卷看起来是一个单独的450MB的卷。但是通过把数据分散到各个磁盘，卷管理器可以优化在条带卷上数据的存取时间。卷管理器对磁盘物理扇区的访问时，就好像它们在各个磁盘的条状分区上被排成了一个序列。如图4-37所示。

因为每一条都是相当较窄的64KB，数据能够被平均分配到每个磁盘上。分条机制增加了数

据读写操作分散在不同盘上的可能性。由于三个盘上的数据可以被同时访问，所以缩短了磁盘I/O的延迟时间，尤其是在负载较重的系统中。

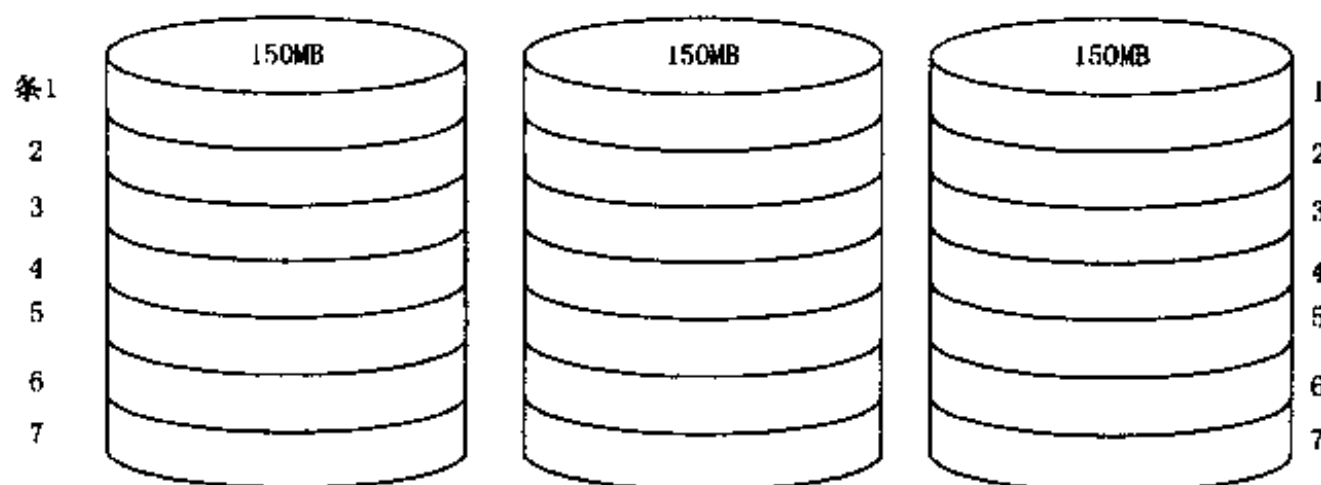


图4-36 有三个盘的条带卷

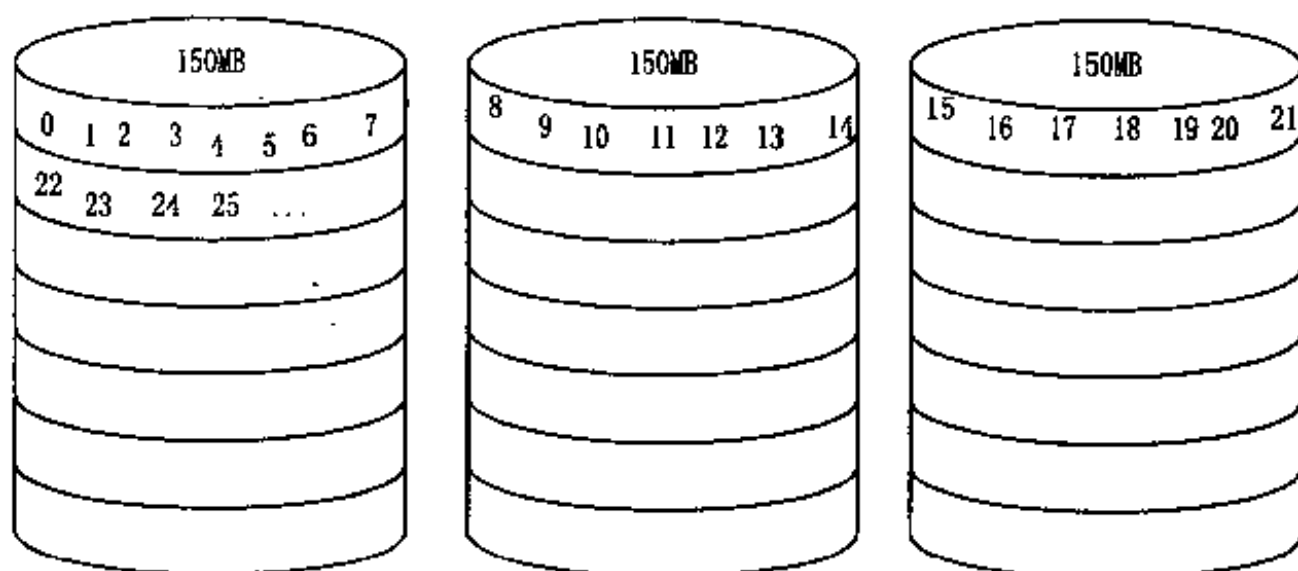


图4-37 条带卷结构

多分区卷使磁盘卷的管理更加方便，而条带卷把磁盘I/O负荷分散到多个磁盘上。但是，这两种卷管理缺点是没有数据的恢复功能。为了恢复数据，卷管理器实现了三种冗余管理机制：镜像卷、RAID-5卷、扇区备用（sector sparing）（在NTFS中介绍）。

3. 镜像卷

在镜像卷中，一个磁盘上分区的内容被复制另一个磁盘与它等大小的分区中。镜像卷有时也被称为RAID-1。镜像卷如图4-38所示。

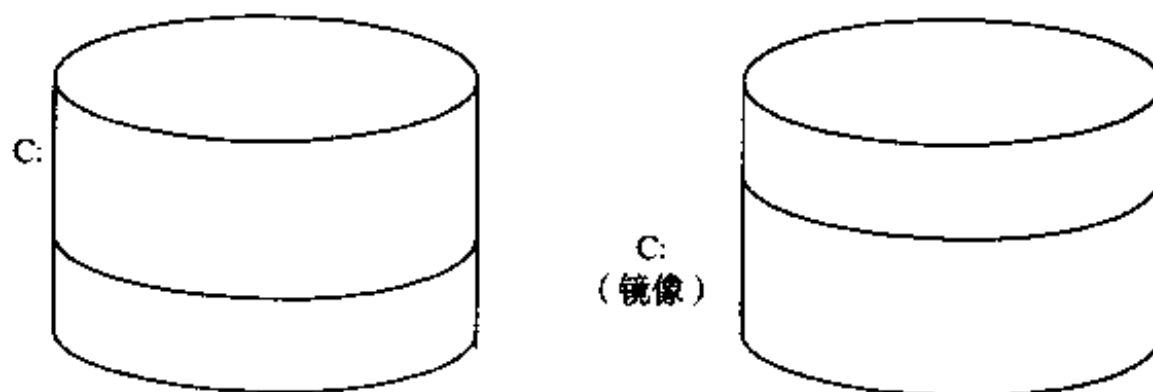


图4-38 镜像卷

当程序向驱动器C:写数据时,卷管理器把同样的数据写在镜像分区的相同位置。如果第一个磁盘或者C:中的任何数据由于硬件或者软件的原因变得不可读时,卷管理器自动从镜像分区中访问数据。镜像卷可以被格式化为任何Windows 2000/XP所支持的文件系统。而文件系统的驱动程序可以保持独立而且不会被卷管理器的镜像活动而干扰。

镜像卷能够在负载很重的系统中可以提高系统的I/O吞吐量。当I/O活动很多时,卷管理器可以在主分区和镜像分区之间平衡I/O操作。两个读操作可以同时进行,所以理论上只用一半时间就可以完成。当修改一个文件时,必须写入镜像卷的两个分区,但是磁盘写操作可以异步进行,所以用户态程序的性能一般不会被这种额外的磁盘更新所影响。

镜像卷是唯一一种支持系统卷和引导卷的多分区卷。原因是Windows 2000/XP的引导代码,包括主引导记录和Ntldr,不具备理解多分区卷的复杂性。镜像卷是个例外,因为引导代码只把它们看作是简单卷,在MS-DOS分区表中只读取被标志为引导或者系统驱动的那一半镜像。由于引导代码并不修改磁盘,它可安全地忽略掉镜像卷的另一半。

4. RAID-5卷

RAID-5卷是普通条带卷的容错变形。RAID-5卷实现了RAID的第五级。它也被称为带奇偶校验的条带卷。容错是通过让一个盘来保存奇偶校验信息来实现的。图 4-39是RAID-5卷的可视化表示。

在图4-39中,第一条的奇偶校验信息存储在盘1中,它包含一个盘2和盘3上第一条的按字节异或(XOR)。第二条的奇偶校验信息保存在盘2中。第三条的奇偶信息保存在盘3中。以这种方式跨磁盘轮流保存奇偶校验信息是一种I/O优化技术。每次向磁盘写数据时,修改字节的奇偶校验信息必须重新计算并重新写入。如果奇偶信息总写入相同的磁盘,那么这个磁盘将成为I/O的瓶颈。

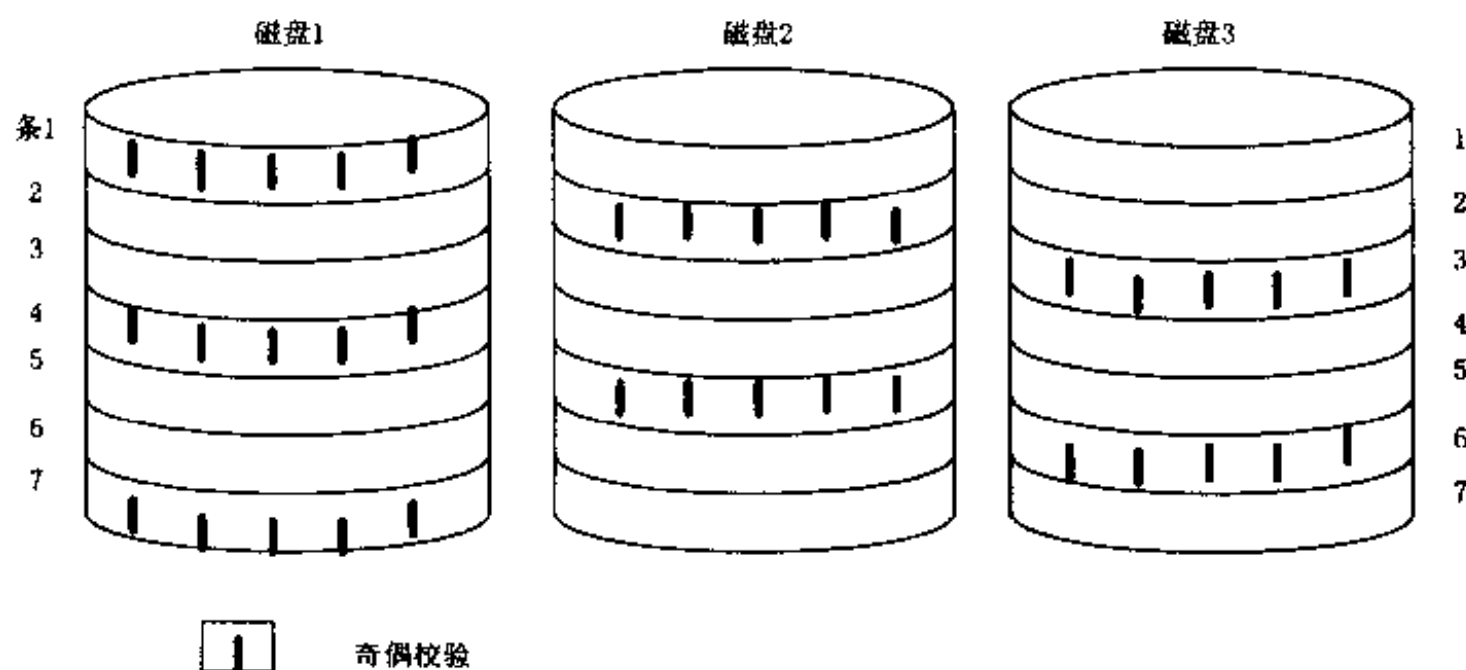


图4-39 RAID-5卷

如果RAID-5卷中的一个盘失效,或者其中一个盘上的信息变得不可读取,卷管理器使用异或操作来重建丢失的数据。要创建一个RAID-5卷,至少要三个硬盘(确切的说,是三个盘上的三个一样大小的分区)。

5. 卷的I/O操作

文件系统驱动程序管理存在卷上的数据，但是它只依靠卷管理器与存储管理驱动程序交互来与卷所在的磁盘交换数据。文件系统驱动程序通过安装过程（后面将有所描述）来获得对卷管理器的卷对象的引用，然后通过这个卷对象向卷管理器发出请求。如果应用程序想直接操纵卷上的数据，也可以不通过文件系统直接向卷管理器发出请求。文件删除操作的恢复程序就是这样的，Windows 2000/XP资源管理工具中DiskProbe也是这样一个程序。

当文件系统或者应用程序对代表卷的设备对象发出I/O请求时，Windows 2000/XP I/O管理器把这个请求（形式为IRP）交给创建设备对象的卷管理器。所以，如果应用程序想要读取系统中第二个卷（假设为简单卷）引导扇区的内容，它可以打开名为\Device\HarddiskVolume2的设备对象，然后对其发出读取从零开始的512字节的请求。I/O管理器把应用程序的请求交给拥有这个设备对象的卷管理器，并通知它这个IRP包的目标是 HarddiskVolume2的设备。

为了逻辑上的方便，Windows 2000/XP用分区来代表物理盘上连续区域，因此卷管理器必须把相对于分区的偏移量转化为相对于磁盘的偏移量。如果分区2从磁盘的3449扇区开始，卷管理器将I/O请求传给磁盘类驱动程序之前，需要用这个值来修改IRP中表示偏移量的参数。磁盘类驱动程序调用小端口驱动程序来完成物理盘的I/O，把所需数据读入应用程序IRP指定的缓冲区中。

4.3.5 卷名字空间

驱动器名分配是外存管理的一个方面，从Windows NT 4到Windows 2000/XP有很大的变化。即使这样，Windows 2000/XP也支持Windows NT 4安装时分配的驱动器名升级为Windows 2000/XP驱动器名。Windows NT 4的驱动器名分配储存在HKLM\SYSTEM\Disk中。Windows 2000/XP在升级过程阅读其中的信息，并存储在系统特定的位置后，将不再访问磁盘键。

1. 安装管理器

安装管理器（Mountmgr.sys）是Windows 2000/XP中新的驱动程序，为在Windows 2000/XP安装后创建的动态磁盘卷和基本磁盘卷分配驱动器名。Windows 2000/XP在HKLM\SYSTEM\MountedDevices中存储所有的驱动器名分配信息。如果按这个键查看注册信息，将会看到以下信息：??\Volume{X}(X是一个GUID)，以及值：??\C:。每个卷都有一个卷名入口，但是一个卷并不一定被分配卷名。

基本磁盘卷的驱动器名和卷名在注册表中存储的数据是Windows NT 4风格的磁盘标记和这些卷第一块分区的起始偏移量。动态磁盘卷在注册表中存储的数据包括卷的DMIO内部GUID。在引导过程中安装管理器进行初始化时，它注册到Windows 2000/XP即插即用子系统中，这样无论是FtDisk还是DMIO创建卷时都将通知它。当安装管理器接到通知时，它确定一个新的卷GUID或者磁盘标记，然后让FtDisk或者DMIO（卷创建者）提供驱动器名分配的建议。FtDisk不会返回建议，而DMIO将在卷数据库入口处查看驱动器名线索。

假如没有推荐的卷驱动器名存在，安装管理器使用卷GUID（或者标识）在内部数据库中进行查询，这个数据库可以反映出注册键的内容。然后安装管理器判断内部数据库是否包含了已分配驱动器名。假如没有包含，安装管理器使用第一个未分配的驱动器名（假如存在一个），定义

一个新的分配，为这次分配创建一个符号链接（例如，\??\D:），并更新MountedDevices 注册表键。如果没有一个可用的驱动器名，就不会进行新驱动器名的分配。同时，安装管理器创建一个卷符号链接（也就是：\??\Volume{X}），并定义一个新的卷GUID。这个GUID与DMIO内部使用的卷GUID不同。

2. 安装点

安装点是Windows 2000/XP的新机制。我们可以将其他卷链接到NTFS卷上的目录，这样可以访问没有驱动器名称的卷。例如，我们可以创建一个名为C:\Project 的NTFS目录，然后在它上面安装一个包含了我们工程目录和文件的卷（NTFS或FAT）。假如工程卷中有一个命名为CurrentProject\Description.txt的文件，我们可以通过目录C:\Projects\CurrentProject\Description.txt访问这个文件。实现安装点的技术是再解析点(Reparse Point)技术。

再解析点是带有固定数据头的任意数据块，Windows 2000/XP将这些数据块与NTFS文件或目录相连接。应用程序或者系统定义再解析点的形式和行为，包括唯一的再解析点标识值，这个值可以识别再解析点属于的应用程序或系统，并说明再解析点数据部分的大小和意义。（数据部分可以有16 KB。）再解析点在一个特定的段中保存了它们唯一的标识。任何实现再解析点的应用程序必须提供一个文件系统过滤器驱动程序，以监视在NTFS卷上进行文件操作时Reparse相关返回代码，并且驱动程序发现返回代码时必须能执行相应的动作。只要NTFS处理文件操作并且遇到带有再解析点文件或者目录时，都会返回Reparse状态码。

Windows 2000/XP NTFS文件系统驱动程序、I/O管理器和目标管理器都部分实现了再解析点功能。通过I/O管理器与文件系统驱动程序的交互，对象管理器初始化路径名解析操作。因此，如果I/O管理器返回一个再解析点状态码，对象管理器必须重新操作。I/O管理器实现了安装点和其他再解析点可能需要的路径名更改功能。NTFS文件系统驱动器必须能够将再解析点数据和文件、路径联系起来，并将它们视为一体。因此，可以将I/O管理器当做是再解析点文件系统过滤器驱动程序。

如果I/O管理器从NTFS得到一个再解析点状态码，而且NTFS返回状态码的文件或者目录没有与Windows 2000/XP内置的再解析点联系起来，同时没有过滤器驱动程序声明这个再解析点，I/O管理器就给对象管理器返回一个出错代码，告知访问文件或者目录的应用程序“系统不可访问这个文件”。

安装点是把卷名（\??\Volume{X}）当做数据信息的再解析点。当使用MMC 磁盘管理工具为卷分配或删除路径时，就在创建安装点。也可以使用内置的命令行工具Mountvol.exe（\Winnt\System32\Mountvol.exe）来创建和显示安装点。

3. 卷安装

Windows 2000/XP为分区分配驱动器名并不意味着分区中包含Windows 2000/XP所识别的文件系统格式的数据。卷的识别过程就是文件系统声明对分区所有权的过程。内核、设备驱动程序、应用程序第一次访问分区上的文件或目录时，发生这个过程。当一个文件系统驱动程序声明它对某个分区负责时，I/O管理器把所有对这个分区的IRP都交给该驱动程序。Windows 2000/XP中的安装操作由三个部分组成：文件系统驱动程序的注册、卷参数块（VPB）和安装请求。

I/O管理器监管安装过程，由于每个文件系统驱动程序在初始化时都向I/O管理器注册，所以它清楚可用的文件系统驱动程序。I/O管理器为本地磁盘（不是网络）文件系统驱动程序提供IoRegisterFileSystem函数进行注册。当一个文件系统驱动程序注册时，I/O管理器保存一个对它的引用，以备I/O管理器在以后的安装操作中使用。

每个设备对象包含一个VPB，但是I/O管理器认为只有卷设备对象的VPB是有意义的。VPB作为卷设备对象和卷安装的文件系统实例设备对象（由文件系统驱动程序创建）之间的纽带。如果VPB的文件系统引用是空的，那么表示卷上没有安装文件系统。I/O管理器打开卷设备对象上的文件或目录时，总要检查卷设备对象的VPB。

例如，安装管理器把D: 分配给系统中的第二个卷，它产生符号连接\??\D:指向设备对象\Device\HarddiskVolume2。一个Win32应用程序试图打开D:上的文件\Temp\Test.txt时，它将会指定路径D:\Temp\Test.txt。Win32子系统在调用NtCreateFile之前将路径转化为\??\D:\Temp\Test.txt。NtCreateFile利用对象管理器来解析这个名字。对象管理器首先碰到遇到了带有路径名\Temp\Test.txt（还没有被解析）的设备对象\Device\HarddiskVolume2。这时，I/O管理器将检查\Device\HarddiskVolume2的VPB是否引用一个文件系统。如果没有，I/O管理器通过一个安装请求来询问每个注册的文件系统驱动程序，是否识别该卷的格式。如果一个文件系统驱动程序作出肯定的回答，I/O管理器将填写VPB，并将一个带着其余路径(\Temp)的打开请求传给该文件系统驱动程序。文件系统驱动程序将完成这个请求，并用自己的文件系统格式来解释分区中存储的数据。如果没有文件系统驱动程序声明对分区的所有权，Windows 2000/XP内置的文件系统驱动程序RAW，将声明对这个分区的所有权，并且拒绝所有对这个分区的打开操作。

4.4 Windows 2000/XP高速缓存管理

Microsoft Windows 2000/XP 高速缓存管理器是一组核心态的函数和系统线程，它们与内存管理器一起为所有Windows 2000/XP文件系统驱动程序提供数据高速缓存（包括本地与网络）。在这一节，将说明Windows 2000/XP高速缓存管理器，包括它的关键内部数据结构和函数是如何工作的，在系统初始化时如何设置，与操作系统中其他元素如何相互作用，以及在Win32 CreateFile中影响文件高速缓存的五个标志。

Windows 2000/XP高速缓存管理器有以下几个主要特征：

1. 单一集中式系统高速缓存

一些操作系统依靠每个单独的文件系统去缓存数据，结果使缓存和内存管理的代码增倍，或者限制了可被缓存的数据的种类。相比之下，Windows 2000/XP提供了一个集中的高速缓存工具来缓存所有的外部存储数据，包括在本地硬盘、软盘、网络文件服务器或是CD-ROM上的数据。任何数据都能被高速缓存，无论它是用户数据流（文件内容和在这个文件上正在进行读和写的活动）或是文件系统的元数据（metadata）（例如目录和文件头）。Windows 2000/XP访问缓存的方法是由被缓存的数据的类型所决定的。

2. 与内存管理器结合

Windows 2000/XP高速缓存管理器一个不寻常的方面是，它从不清楚在物理内存中有多少缓

存数据。这听起来可能有些奇怪，因为高速缓存目的是通过在物理内存中保留经常存取的数据的一个子集来改善I/O性能。而Windows 2000/XP高速缓存管理器不知道多少数据存在物理内存，因为它采用将文件视图映射到系统虚拟空间的方法访问数据，在这过程中使用了标准区域对象（section object）。访问位于映射视图中的地址时，内存管理器不在物理内存的逻辑块中分配页面。以后需要内存时，内存管理器再将高速缓存中的数据页面换出，写回映射文件。

通过映射文件实现基于虚拟地址空间的高速缓存，高速缓存管理器在访问缓存中文件的数据时避免产生读写I/O请求包（IRP）。取而代之，它仅仅在内存和被缓存的文件部分所被映射的虚拟地址之间拷贝数据，并依靠内存管理器去处理换页。（高速缓存管理器也初使化I/O，如延迟写，将在本章后面详述。）这种设计使打开缓存文件就像将文件映射到用户地址空间一样。

3. 高速缓存的一致性

高速缓存管理器一个重要的功能是保证任何访问高速缓存数据的进程可得到这些数据的最新版本。当进程打开一个文件（这个文件被缓存了）而另一个进程直接将文件映射到它的地址空间（运用Win32 MapViewOfFile函数），问题就产生了。这种潜在的问题不会在Windows 2000/XP中出现，因为高速缓存管理器和用户应用程序使用相同的内存管理文件映射服务将文件映射到它们的地址空间。而内存管理器保证每一个被映射文件只有唯一的代表，它映射文件的所有视图（即使它们相互重叠）到物理内存页面的单独集合，如图4-40所示。

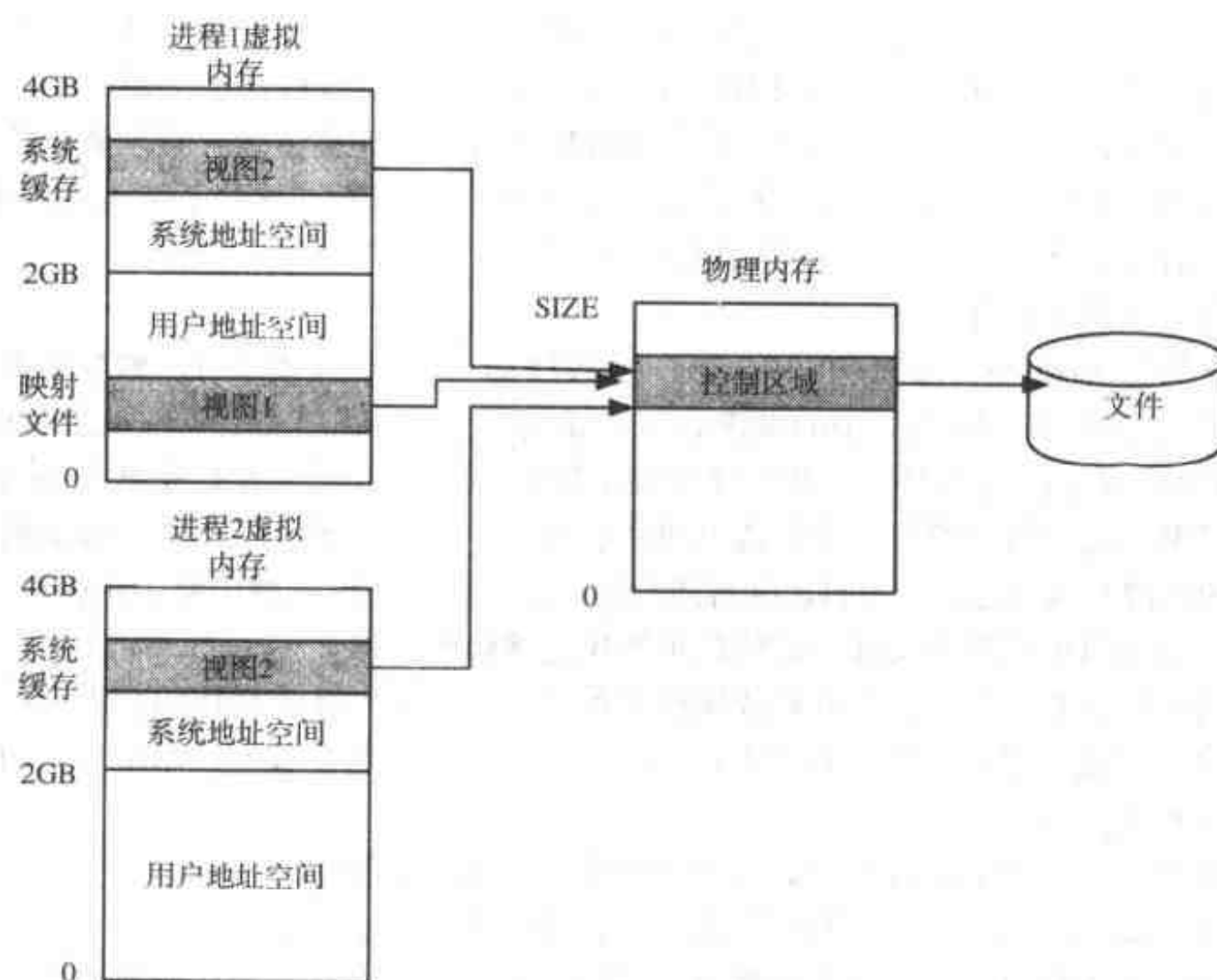


图4-40 一致的缓存架构

例如，如果进程1有一个文件视图被映射到其用户地址空间，进程2通过系统缓存访问同一视

图。进程1正在做的任何改变，进程2都可以看到，而不用等到这些改变被回写。

4. 虚拟块缓存

大多数操作系统高速缓存管理器（包括Novell NetWare, OpenVMS, OS/2和老的UNIX系统）基于磁盘逻辑块(logical block)缓存数据。用这种方式，高速缓存管理器知道磁盘分区中的哪些块在高速缓存中。与之相比，Windows 2000/XP高速缓存管理器用一种虚拟块缓存(virtual block caching)方式，管理器对缓存中文件的某些部分进行追踪。通过内存管理器的特殊系统高速缓存例程将256KB大小的文件视图映射到系统虚拟地址空间，高速缓存管理器能够管理文件的这些部分。这种方式有以下几个主要特点：

(1) 它使智能的文件预读成为可能。因为高速缓存能够追踪哪些文件的哪些部分在缓存中，因而能够预测调用者下一步将访问哪里。

(2) 它允许I/O系统绕开文件系统访问已经在缓存中的数据（快速I/O）。因为高速缓存管理器知道哪些文件的哪些部分在缓存中，它能返回被缓存数据的地址满足I/O的需要，而不调用文件系统。

关于如何预读和快速I/O将在后面详细讲解。

5. 基于流的缓存

Windows 2000/XP高速缓存管理器与文件缓存相对应也设计了字节流的缓存。一个流是指在文件内的字节序列。一些文件系统，像NTFS，允许文件包括多个流对象。高速缓存管理器通过独立地缓存每一个字节流来适应这些文件系统。NTFS能够拥有这种特点，得益于把主文件表放入字节流中并缓存这些字节流。事实上，虽然Windows 2000/XP高速缓存管理器被认为是高速缓存文件，但它实际上缓存的是字节流（所有文件至少有一数据流）。这些字节流通过文件名标识，如果在文件中存在多个字节流，还要标明字节流名。

6. 可恢复文件系统支持

可恢复文件系统(recoverable file system)，如NTFS，在系统失败后可以修复磁盘卷结构。这就是说，当系统失败时正在进行的I/O操作必须全部完成，或在系统重启动时从磁盘中全部恢复。未完成的I/O操作可能破坏磁盘卷，甚至导致整个磁盘卷不可访问。为了避免这个问题，在改变卷之前，可恢复文件系统会维护一个日志文件(log file)。在每一次涉及文件系统结构（文件系统的元数据）的修改写入卷之前，该日志文件进行记录。如果因系统失败中断了正在进行的卷修改，可恢复文件系统可以根据日志文件中的信息重新执行卷修改操作。

为保证成功地恢复一个卷，在卷修改操作开始之前，记录卷修改操作的日志记录必须被完全写入磁盘。由于写磁盘操作可以被高速缓存，因此高速缓存管理器和文件系统必须协同工作以确保下列操作按顺序进行：

- 1) 文件系统写一个日志文件记录，记录将要进行的卷修改操作。
- 2) 文件系统调用高速缓存管理器将日志文件记录刷新到磁盘上。
- 3) 文件系统把卷修改内容写入高速缓存，即修改文件系统在高速缓存的元数据。
- 4) 高速缓存管理器将被更改的元数据刷新到磁盘上，更新卷结构。（实际上，与卷修改一样，日志文件记录经过批处理之后才刷新到磁盘上。）

4.4.1 高速缓存的结构

因为Windows 2000/XP系统高速缓存管理器基于虚拟空间缓存数据，所以它管理一块系统虚拟地址空间区域（而不是一块物理内存区域）。高速缓存管理器把每个地址空间区域分成256KB的槽(slot)，被称为视图(view)，如图4-41所示。

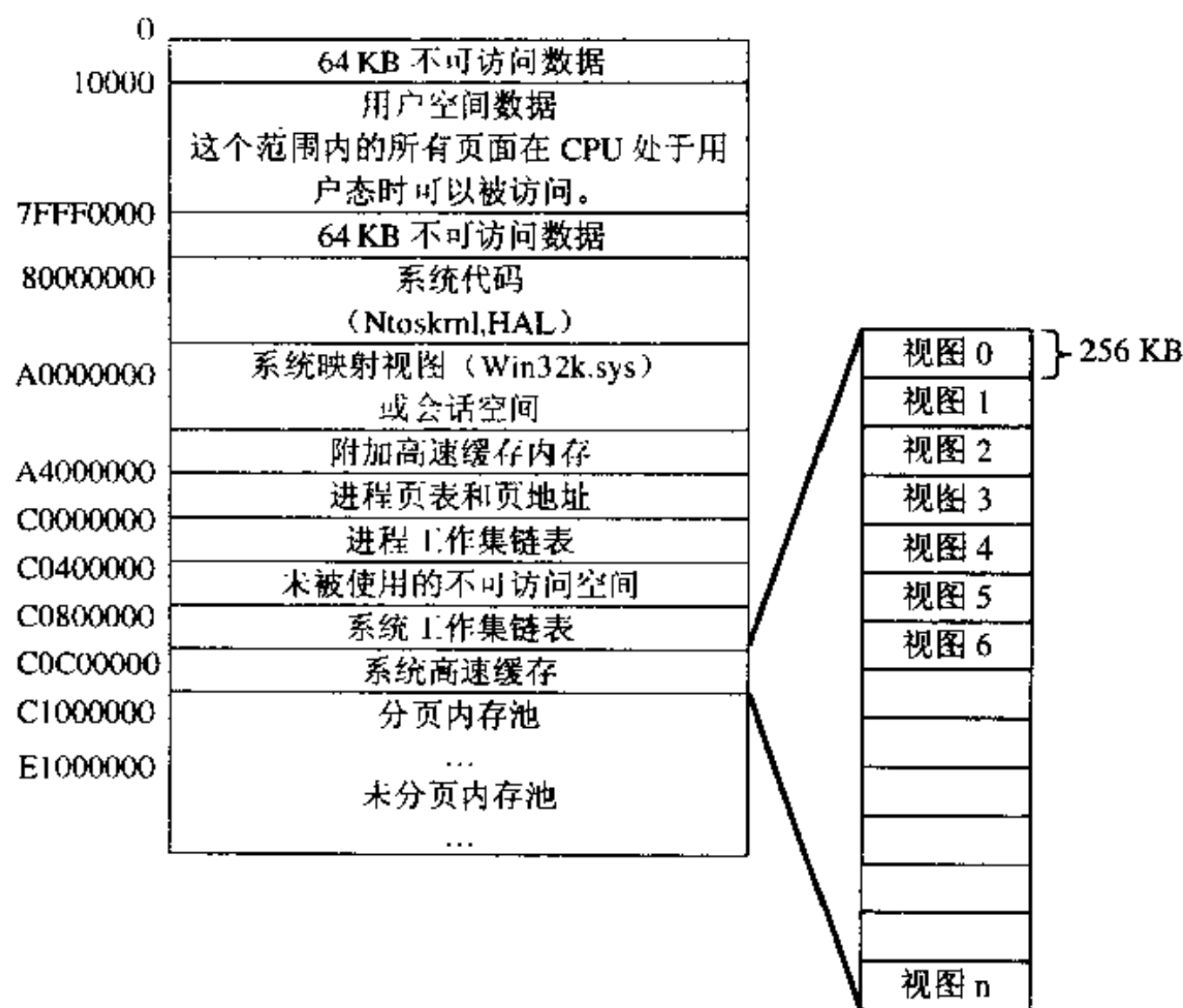


图4-41 系统高速缓存地址空间

文件第一次I/O（读或写）操作时，高速缓存管理器将文件中包含被请求数据的256 KB对齐的区域映射为一个256 KB视图，放入到系统缓存空间的一个空闲槽内。例如，如果从偏移量为300 000字区域处开始读入10字节数据，被映射的视图将在偏移量262144处开始（文件第二个256 KB对齐区域）容量为256 KB。

高速缓存管理器在文件视图和缓存地址空间的槽之间循环进行映射，将所请求的第一个视图映射到第一个256 KB槽中，再将第二个视图映射到第二个256 KB槽中，以此类推，如图4-42所示。在这个例子中文件B最先被映射，文件A其次，文件C再次，所以文件B被映射的块占据高速缓存的第一个槽。注意文件B中只有第一个256 KB部分被映射，这是由于这个文件只有该部分被访问。虽然文件C只有100 KB（比系统缓存视图要小），但它仍在缓存中占据独立的256 KB槽。

高速缓存管理器只映射活跃的视图。然而只有在读或写文件操作时，视图被标记为活跃。除非进程用带有FILE-FLAG-RANDOM-ACCESS标志的CreatFile函数打开一个文件，否则高速缓存

管理器在映射文件新视图时，不映射那些未被激活的文件视图。

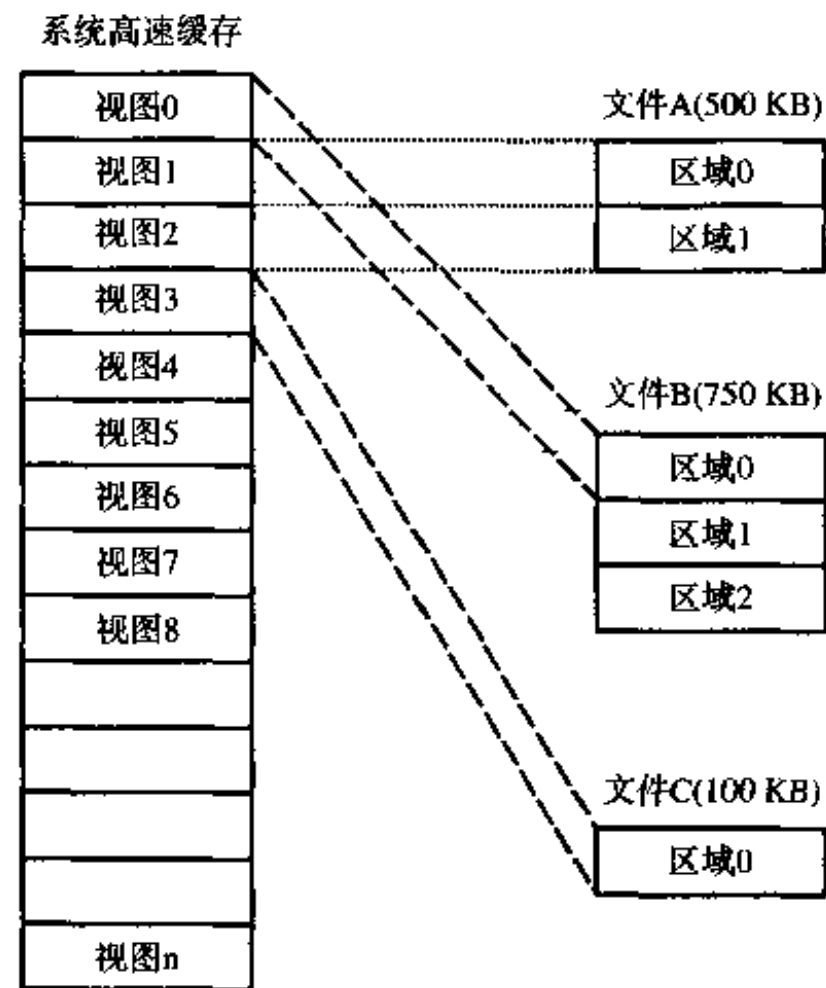


图4-42 映射到系统高速缓存中的不同大小文件

当高速缓存管理器需要映射一个文件视图但缓存内没有空余的槽时，它将取消最近一个未激活映射视图，并使用这个槽。如果没有视图可用，则返回一个I/O错误，说明没有足够的系统资源完成操作。然而，由于只有在进行读或写操作的视图才被激活，上述情况只有在成千上万的文件被同时访问时才会发生。

4.4.2 高速缓存的大小

接下来我们将解释Windows 2000/XP是如何计算系统高速缓存的大小的（包括虚拟与物理的高速缓存）。正如大多数与内存管理相关的计算一样，系统缓存的大小依赖于包括内存大小和运行的Windows 2000/XP版本等因素。

1. 缓存区的虚拟大小

系统高速缓存虚拟大小是已安装物理内存总量的函数，默认大小为64 MB。如果系统物理内存多于4032页（16 MB），缓存大小设定为以128 MB为基础，物理内存每比16 MB多4 MB，则增加64 MB缓存区。利用这种算法，有64 MB物理内存的计算机系统虚拟缓存将是：

$$128\text{ MB} + (64\text{ MB} - 16\text{ MB}) / 4\text{ MB} * 64\text{ MB} \approx 896\text{ MB}$$

表4-22列出了系统缓存最小和最大的虚拟大小，还有开始与结束地址。如果系统计算出虚拟大小大于512 MB，缓存就被赋予额外的地址区域，称为缓存附加内存。表4-23列出了记录缓存虚拟大小和系统高速缓存地址系统变量。

表4-22 系统数据高速缓存的容量和位置

平 台	地 址 范 围	最大/最小虚拟容量
x86 2GB系统空间	0xC1000000-E0FFFFFF, 0xA4000000-BFFFFFFF	64 / 960 MB
x86 1GB系统空间	0xC1000000-DBFFFFFF	64 / 432 MB
x86 1GB系统空间 有终端服务	0xC1000000-DCFFFFFF	64 / 448 MB

表4-23 记录系统高速缓存虚拟大小和地址的系统变量

系 统 变 量	描 述
MmSystemCacheStart	高速缓存的起始虚拟地址
MmSystemCacheEnd	高速缓存的终止虚拟地址
MiSystemCacheStartExtra	容量>512MB的高速缓存额外部分的起始虚拟地址
MiSystemCacheEndExtra	容量>512MB的高速缓存额外部分的终止虚拟地址
MmSizeOfSystemCacheInPages	以页计算的高速缓存最大容量

2. 缓存的物理大小

前面曾提到过，Windows 2000/XP的高速缓存与其他操作系统设计上最大不同是由全局内存管理器来管理物理内存。正因为这样，用来处理工作集的扩展和收缩、管理已修改和未修改链表的代码也被用来控制系统缓存的大小，并动态地平衡进程和操作系统间对物理内存的需求。

系统高速缓存没有自己的工作集，而是与高速缓存数据、页缓冲池、可分页的核心代码以及可分页的驱动程序代码共用一个系统工作集。尽管系统高速缓存只是这个工作集的一个组成部分，在系统内部将它称为“系统高速缓存工作集”。在本书中我们将它简称为系统工作集。

通过观察性能计数器和系统变量，我们可以对比系统高速缓存的物理大小与整个系统工作集的物理大小，同样，也能看到系统工作集上的缺页面信息，如表4-24所示。

表4-24 记录系统高速缓存的物理容量和缺页信息的系统变量

性能计数器（以字节为单位）	系统变量（以页为单位）	描 述
Memory: System Cache Resident Bytes	MmSystemCachePage	系统高速缓存消耗的物理内存
Memory: Cache Bytes	MmSystemCachesWs.WorkingSetSize	系统工作集的总容量（包括高速缓存、页池、可换页代码和系统映射视图）。这不是高速缓存的容量（字面含义）
Memory: Cache Bytes Peak	MmSystemCacheWs.Peak	系统工作集容量的峰值
Memory: Cache Faults/Sec	MmSystemCacheWs.Page-FaultCount	系统工作集中的缺页（不仅使高速缓存）

4.4.3 高速缓存的数据结构

高速缓存管理器利用下面的数据结构跟踪被缓存的文件。

- 1) 在系统高速缓存的每个256 KB的槽由一个VACB描述。
- 2) 每个打开的被缓存文件有一个专用的缓存映射，它包含了用于控制文件预读的信息。
- 3) 每个被缓存的文件有一个单独的共享缓存映射结构，它指向系统缓存中包含此文件映射视图的槽。

这些结构及它们的关系将在下一节描述。

1. 系统范围的高速缓存数据结构

高速缓存管理器通过用虚拟地址控制块（virtual address control block, VACB）在系统缓存中追踪视图的状态。在系统初始化期间，高速缓存管理器分配一个单独的未分页的内存区域来保存所有用来描述系统高速缓存的VACB。在变量CcVacbs中存储VACB数组的地址。每个VACB代表系统高速缓存中一个256 KB的视图，如图4-43所示。VACB结构如图4-44所示。

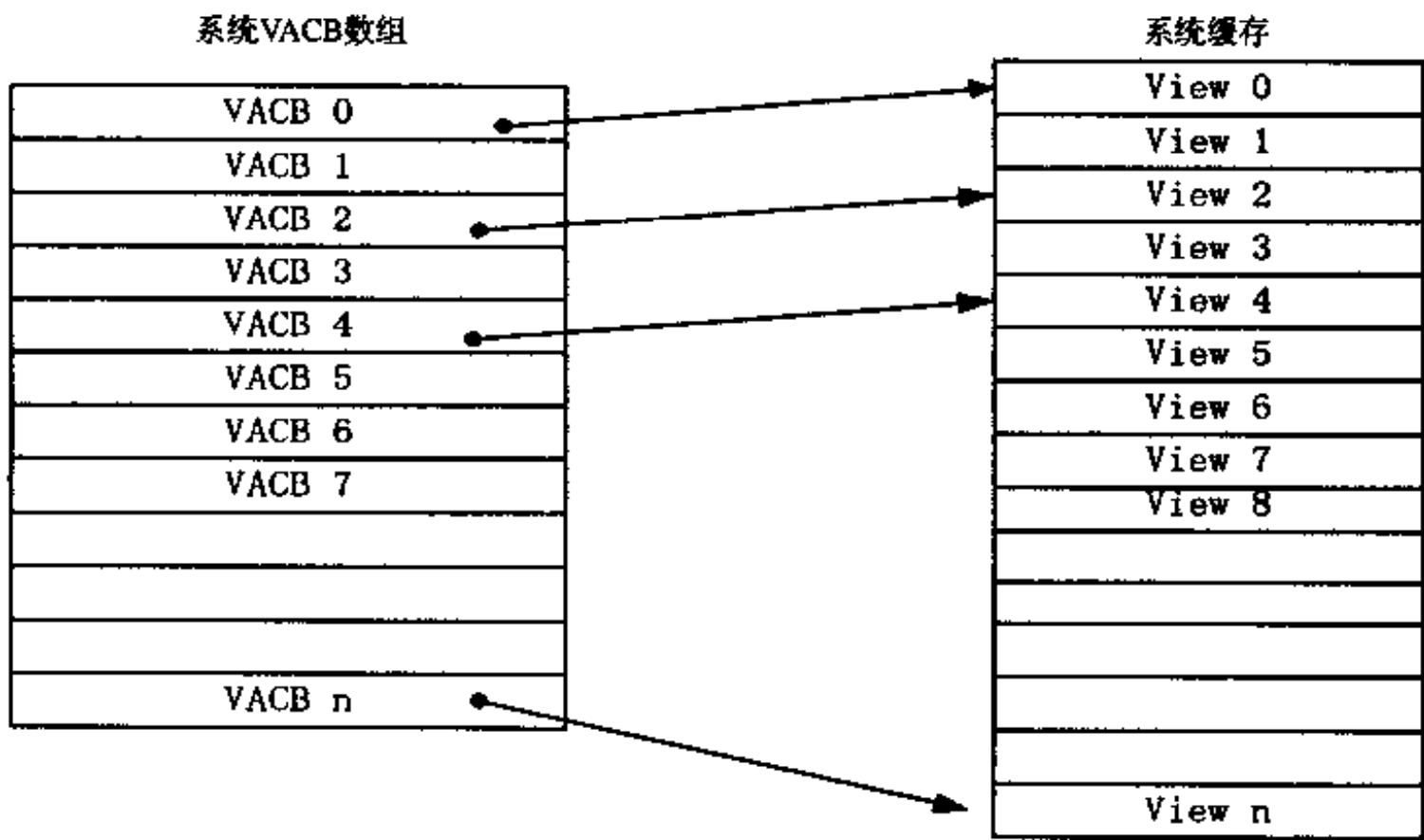


图4-43 系统VACB数组

从图4-44中可以看到，VACB中的第一个字段存放系统高速缓存中数据的虚拟地址。第二个字段存放被共享缓存映射结构的指针，它指出哪一个文件被缓存。第三个字段标识视图起始处在文件内的偏移（通常在256KB间隔基础上建立）。最后一个字段，VACB包括引用视图的数目，即有多少读或写操作正在访问该视图。在一个文件进行I/O操作时，文件的VACB引用

系统高速缓存中数据的虚拟地址
指向共享高速缓存映射的指针
文件偏移
活动计数

图4-44 VACB结构

第二个入口指第二个256 KB，以此类推。图4-46显示了三个被映射到系统缓存中的不同文件的四个不同部分的情况。

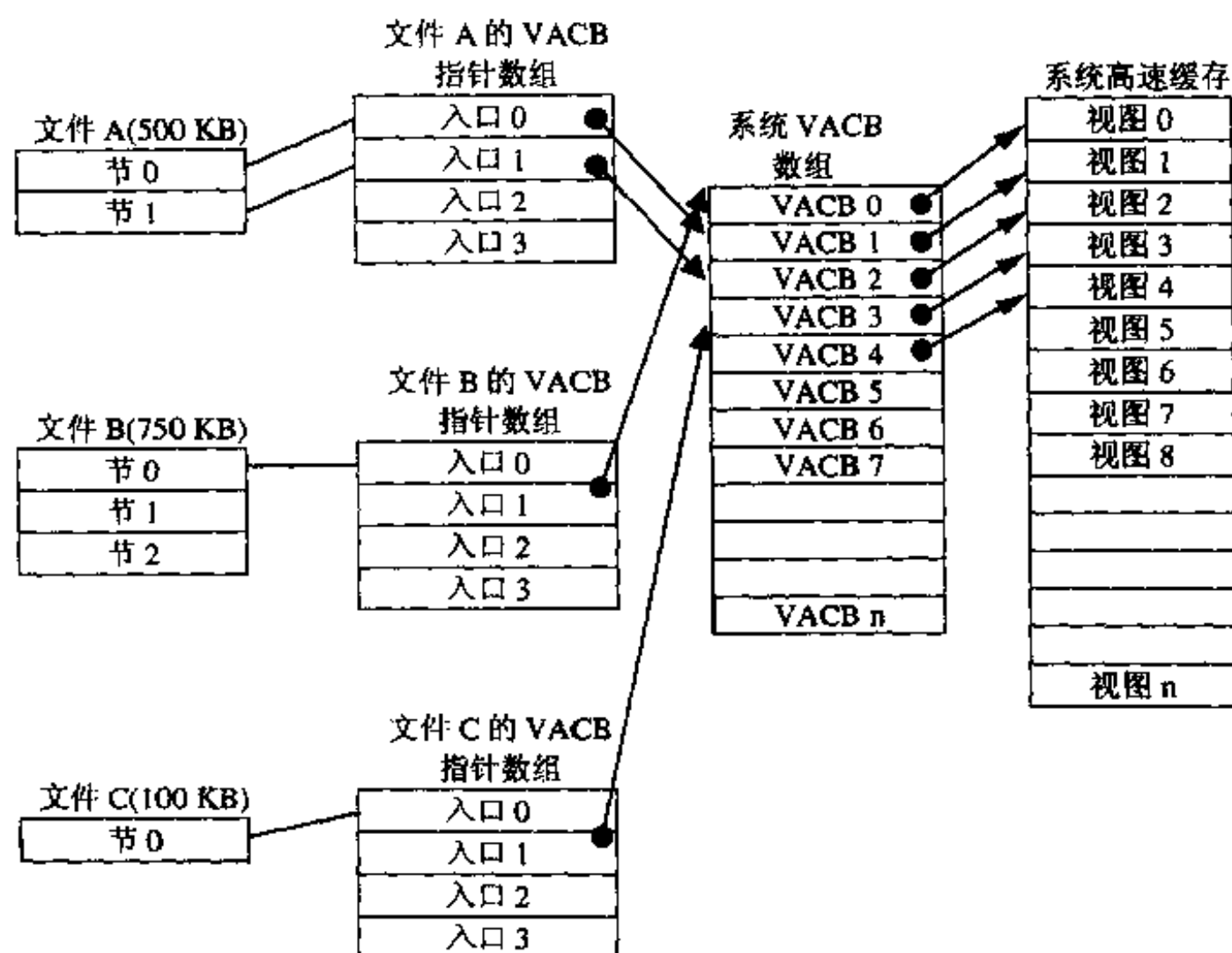


图4-46 VACB 指针数组

当进程访问一个文件的指定位置时，高速缓存管理器在文件的VACB指针数组中查找相应的入口，看被请求的数据是否已经被映射到了缓存中。如果数组的那个入口不为空（因此含有一个指向VACB的指针），则文件中的那一部分已经在缓存中。VACB会依次指向文件视图被映射到系统高速缓存中的位置。如果该项为空，则高速缓存管理器必须在系统缓存区中找到一个空闲的槽（还有一个空闲的VACB）来映射所请求的视图。

出于空间优化的考虑，共享高速缓存映射包含一个有4项的VACB指针数组。因为每个VACB描述256 KB，这个固定大小的数组最多可以描述1 MB的文件。如果文件超过1 MB，则从未分页内存池中分配一个单独的VACB指针数组，该数组的大小等于文件的大小除以256 KB（如果有余数，商加1）。然后，共享高速缓存映射指向这个独立的结构。

为进一步进行优化，如果文件超过32 GB大小，从未分页池中分配的VACB指针数组组成多维指针的稀疏数组(sparse multilevel index array)，这时每个指针数组含有128项。用下面的公式，可以计算出文件的指针数组需要多少层：

$$(\text{表示文件大小所需的二进制位数} - 18) / 7$$

将公式的结果取整。公式中的18是由VACB代表256 KB而来的，256 KB等于 2^{18} 。7来源于数

组中的每一级有128项，而 2^7 是128。因此，一个大小为 2^{24} （高速缓存管理器支持的最大数）的文件只需要7层。数组之所以稀疏，是因为被高速缓存管理器分配的分支中只有最下层数组才指向VACB。图4-47显示了一个多维VACB数组的例子，例子中的稀疏矩阵文件需要用三层来表示。

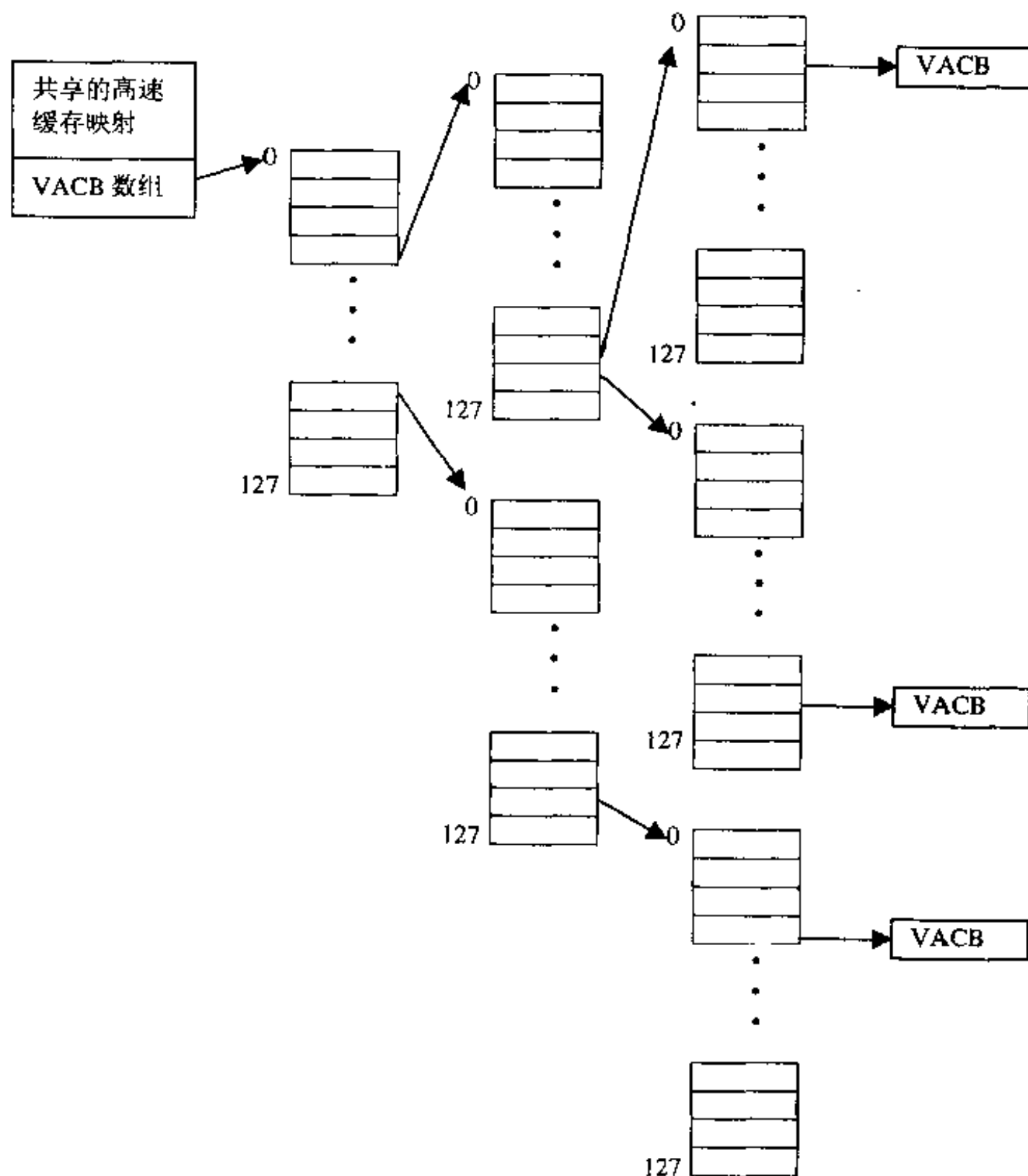


图4-47 多维VACB数组

这种方案处理稀疏文件很有效，这些文件很大但只含有少量所需数据，所以只需分配足够的数组处理当前文件视图就行了。例如，32 GB的稀疏文件只有256 KB被映射到了高速缓存的虚拟地址空间。这只需分配一个VACB和3个指针数组，因为数组只有一个分支被映射了。而这样一个32 GB（ 2^{28} ）的文件只需一个3层的数组。如果高速缓存管理器对这个文件不采用多级VACB数组优化，它将不得不分配一个有128 000项的数组，或相当于分配1000个指针数组。

4.4.4 高速缓存的操作

这一节将介绍高速缓存管理器怎样为文件系统驱动程序实现读写文件数据。记住，在文件I/O过程中仅当文件被打开时（例如，用Win32 CreateFile函数），高速缓存管理器才被激活。被映射的文件，以及用FILE_FLAG_NO_BUFFERING标记打开的文件都不会经过高速缓存管理器。

1. 回写缓存和延迟写

Windows 2000/XP高速缓存管理器实现了一个带有延迟写(lazy writing)的回写(write-back)高速缓存。这意味着写入文件的数据首先被存储在高速缓存页面的内存中，然后再被写入磁盘。因此，写操作允许在短时间内积累，并一次性刷新到磁盘。这可以减少磁盘的I/O次数。

高速缓存管理器必须显式地调用内存管理器去刷新高速缓存页面，如果不这样，内存管理器只在对物理内存的需求超过所提供的内存时，才将内存的内容写入磁盘。这种方式适合于反复变化的数据。然而，被缓存的文件数据也可能不是频繁变化的数据。如果一个进程修改了高速缓存的数据，用户期望将修改过的部分及时地反映到磁盘上。

确定高速缓存的刷新频率是十分重要的。如果高速缓存刷新过于频繁，系统将因为不必要的I/O操作而降低性能。如果高速缓存刷新过少，用户将面临在系统崩溃时失去已修改数据的风险（特别是对于那些已经让程序保存修改数据的用户，这极难接受），或将物理内存用光的危险（因为已修改的页面占用了过多的内存）。

为了平衡这些关系，由高速缓存管理器每秒钟产生一个系统线程——延迟写器——排列系统缓存中1/8的“脏”(dirty)页（已修改页），并将其写入磁盘。如果脏页产生的频率大于延迟写器写入页面的频率，延迟写器将按脏页产生的频率计算出额外需要写入的页面数，将这些页面写入磁盘。

通过观察表4-25中列出的高速缓存性能计数器或系统变量，可以查看延迟写器的活动。

表4-25 用于检查延迟写活动的系统变量

系统计数器（频度）	系统变量（数值）	描 述
Cache: Lazy Write Flushes/Sec	CcLazyWriteIos	延迟写器刷新的次数
Cache: Lazy Write Pages/Sec	CcLazyWritePages	被延迟写器写入的页面数量

2. 计算脏页阈值

脏页阈值(threshold)是系统唤醒延迟写系统线程将页面写回到磁盘之前，保存在内存中的系统高速缓存的页面数。该数值在系统初始化时计算，且依赖于物理内存大小和注册表项HKLM\SYSTEM\CurrentControlSet\Control\SessionManager\MemoryManagement\LargeSystemCache。在Windows 2000/XP Professional 中这个值缺省是0，在Windows 2000/XP Server中缺省是1。可以在Windows 2000/XP Server系统图形界面中通过修改文件服务属性来调整这个值。（打开网络连接的属性，双击Microsoft网络文件和打印共享。）尽管这项服务也存在于Windows 2000/XP Professional，但它的参数不可以调整。

表4-26给出了用于计算脏页阈值的算法。当系统最大工作集的大小超过4 MB时（经常是这样），表4-26的计算将被忽略，脏页阈值被设置为系统最大工作集大小减去2 MB的页数。

表4-26 计算脏页阈值的算法

系统内存容量	脏页阈值
小	物理页面数 / 8
中	物理页面数 / 4
大	上面两数值的和

3. 屏蔽对文件延迟写

在调用Win32 CreateFile函数时指定FILE_ATTRIBUTE_TEMPORARY标志创建一个临时文件，延迟写器就不会将脏页写回磁盘，除非物理内存严重不足或文件关闭。延迟写器的这种特性改善了系统性能——延迟写器不会立即将最终可能丢弃的数据写入磁盘。应用程序经常在关闭了临时文件后将其删去。

4. 强制写缓存到磁盘

由于一些应用程序不允许在向磁盘写文件和查看磁盘数据更新之间出现即使很短的延迟，所以高速缓存管理器也支持基于单个文件的通写高速缓存，即数据一改变被立即写入磁盘。要启动通写高速缓存，需要在调用CreateFile函数时设置FILE_FLAG_WRITE_THROUGH标志。作为另一种选择，当一个线程需要把数据写入磁盘时，可以使用Win32 FlushFileBuffers函数显式地刷新一个打开的文件。可以借助性能计数器或表4-27中的系统变量观察高速缓存刷新，这些刷新源于通写缓存的I/O请求或显式地调用FlushFileBuffers函数的结果。

表4-27 用于观察缓存刷新操作的系统变量

系统计数器（频度）	系统变量（数值）	描 述
Cache: Data Flushes/Sec	CcDataFlushes	高速缓存页被显式刷新或由于通写而刷新的次数
Cache: Data Flush Pages/Sec	CcDataPages	显式刷新或由于通写而刷新的页面数量

5. 刷新被映射的文件

如果延迟写器必须从映射到其他进程地址空间的视图向磁盘写入数据，情况就有些复杂，因为高速缓存管理器仅知道它修改过的页面。（被其他进程修改的页面只有该进程知道，因为被修改页在页表项中的修改标志被保存在进程私有页表中。）为了处理这种情况，当用户映射一个文件时，内存管理器就会通知高速缓存管理器。当该文件在高速缓存内被刷新时（例如，调用了Win32 FlushFileBuffers函数），高速缓存管理器将缓存中的脏页写入磁盘，然后检查文件是否被其他进程映射。如果文件也被其他进程映射了，那么高速缓存管理器把文件区域所对应的整个视图刷新一遍，以便将第二个进程可能改变的页面写入磁盘。如果用户映射了一个也在高速缓存中打开的视图，当该视图被取消映射时，修改过的页被标记为“脏”以便延迟写线程将来刷新该视图时，将这些脏页写入磁盘。这些过程只有按下列次序进行才能正常起作用：

- 1) 用户取消了视图的映射；
- 2) 进程刷新文件缓冲区。

如果没有遵守这个次序，则无法预测哪些页面会被写入磁盘。

6. 智能预读

Windows 2000/XP 高速缓存管理器运用空间局部性原理，基于进程当前所读取数据预测其下一步可能读的数据，从而实现智能预读(intelligent read-ahead)。因为系统缓存是以虚拟地址为基础，而虚拟地址对于一个文件而言是连续的，它们在物理内存中是否连续并不重要。基于逻辑块的高速缓存系统是以磁盘上被访问的数据的相对位置为基础，而文件未必连续存储在磁盘上。所以对于逻辑块高速缓存的文件预读会更复杂，而且需要文件系统驱动程序和逻辑块高速缓存的紧密配合。

预读有两种类型——虚拟地址预读和带历史信息的异步预读，这两种类型将在下面两段中解释。利用Cache: Read Aheads/Sec 性能计数器或者CcReadAheadIos系统变量可以检查预读的活动。

7. 虚拟地址预读

当内存管理解决缺页时，它会将访问页面相近的几个页一起读到内存中，这种方法叫做簇。对于顺序读的应用程序，这种虚拟地址预读(virtual address read-ahead)操作减少了获取数据所需的磁盘读操作次数。内存管理器的这种方法唯一缺点是：由于这种预读方式是在处理缺页的上下文中进行的，所以它必须同步进行，此时等待页面数据的线程必须处在等待状态。

8. 带历史信息的异步预读

由内存管理器进行的虚拟地址预读提升了系统的I/O性能，但是它只对顺序访问的数据有利。为了将预读的好处扩展到特定的随机访问数据中，高速缓存管理器在文件的私有缓存映射结构中为正在被访问的文件句柄保存最后两次读请求的历史信息，这种方法被称为“带历史信息的异步预读”(asynchronous read-ahead with history)。如果能从调用者明显的随机读取中确定一种模式，高速缓存管理器将这种模式延伸。例如，如果调用者读了第4000页，然后是第3000页，高速缓存管理器假设调用值下一个请求的页面会是第2000页，于是预读第2000页。

为了提高预读效率，Win32 CreateFile函数提供了一个表示顺序文件访问的标志：FILE_FLAG_SEQUENTIAL_SCAN。如果设置了这个标志，高速缓存管理器不会为调用者保存用于预测的历史纪录，而是进行顺序预读。由于文件是被读入了高速缓存的工作集，高速缓存管理器取消映射不再活跃的文件视图时，通知内存管理器将属于这些视图的页面放到备用链表或修改链表中（如果页面被修改过），以便它们可以被再次使用。对每次读取的I/O操作，高速缓存管理器预读的数据是原来的三倍（例如读192KB代替64KB）。随着调用者继续读，高速缓存管理器向前预读附加的数据块，始终保持着比调用者提前读入一个读操作（当前读取数据的大小）。

高速缓存管理器的预读是异步的，由于执行预读的线程与读入数据的线程不同，所以两者可以同时执行。当请求读取被缓存的数据时，高速缓存管理器首先访问被请求的虚页而，并完成这次请求，然后向系统工作线程提出另一次I/O请求，取得额外的数据。接下来，系统工作线程在后台执行，读入预期调用者下一次将请求的数据。当程序继续执行时，预先读的页而已被调入内存，所以调用者再次请求数据时它已经在内存中了。

尽管，带历史信息的异步预读技术比普通预读使用了更多的内存，但它大大改善了程序读大量顺序缓存的数据的性能。Cache: Read Aheads/Sec性能计数器显示了顺序访问预读的操作。

对于无法预测读取模式的应用程序，可以在调用Win32 CreateFile函数时设置FILE_FLAG_

RANDOM_ACCESS标志。这个标志通知高速缓存管理器不要试图预测应用程序下一步要读取的位置，这样就关掉了预读功能。这个标志项还阻止了高速缓存管理器在文件被访问时强制取消其视图映射，这样程序再次访问文件某个部分时减少了映射/取消映射的活动。

9. 系统线程

高速缓存管理器通过向公共临界系统工作线程池发送请求来实现延迟写和预读的I/O操作。然而，可供使用的线程数有限制，对于小型和中型内存的系统，数目比临界工作系统线程的总数少一个（大内存系统少两个）。

在内部，高速缓存管理器将它的工作请求组织到两张表中（尽管是同一组工作线程为这些表服务）：

- 1) 用于预读操作快速队列；
- 2) 用于延迟写扫描（刷新脏页数据）、后台写和延迟关闭的常规队列。

为了追踪工作线程需要进行的工作项目，高速缓存管理器创建了自己内部的处理器后备链表。每个处理器有一个定长的包含工作队列项目结构的表。工作队列项目的数量取决于系统大小：小内存系统为32，中内存系统为64，大内存Windows 2000/XP Professional系统为128，大内存Windows 2000/XP Server系统为256。

10. 快速I/O

任何时候只要有可能，读写被缓存文件可以用被称为快速I/O (fast I/O) 的高速机制来处理。快速I/O读写一个缓存的文件不需要产生I/O请求包（IRP）。有了快速I/O机制，I/O管理器可以调用文件系统驱动程序快速I/O例程来查看是否能够直接从高速缓存管理器得到所需的数据，而不需产生IRP。

由于Windows 2000/XP 高速缓存管理器能够追踪哪些文件的哪些块在高速缓存中，所以文件系统驱动程序能够利用高速缓存管理器通过简单的拷贝那些在高速缓存中的页面来访问数据，而不用产生IRP。

快速I/O并不总是发生。例如，文件的第一次读写时需要设置该文件以供高速缓存（将文件映射到高速缓存中，设置高速缓存数据结构）。如果调用者指定了异步读写，快速I/O也不能使用。因为在满足将缓冲区与系统高速缓存之间的拷贝而进行的换页I/O操作期间，调用者可能被停止，这样就不能真正提供所要求的异步I/O操作。即使在同步I/O操作时，文件系统驱动程序也可能会判定它不能用快速I/O机制，例如，如果正在操作的文件有一个被锁定的字节区域（就像调用Win32 LockFile 和UnlockFile函数的结果）。因为高速缓存管理器不知道哪个文件的哪个部分被上了锁，文件系统驱动程序必须检查读写的合法性，这需要产生IRP。快速I/O的决策树如图4-48所示。

下面是快速I/O服务进行读写操作时涉及的几个步骤：

- 1) 线程进行一个读或写的操作；
- 2) 如果文件被缓存而且I/O同步时，I/O请求就会被传送到文件系统驱动程序快速I/O入口点。如果文件没有被缓存，文件系统驱动程序设置该文件用于高速缓存，这样下一次就可以使用快速I/O来满足读写请求。

- 3) 如果文件系统驱动程序快速I/O例程断定可以使用快速I/O，它就调用高速缓存管理器的

读或写例程去直接访问缓存中的数据。(如果快速I/O不可能进行,文件系统驱动程序返回到I/O系统,之后为I/O产生一个IRP,最终调用文件系统的常规读例程。)

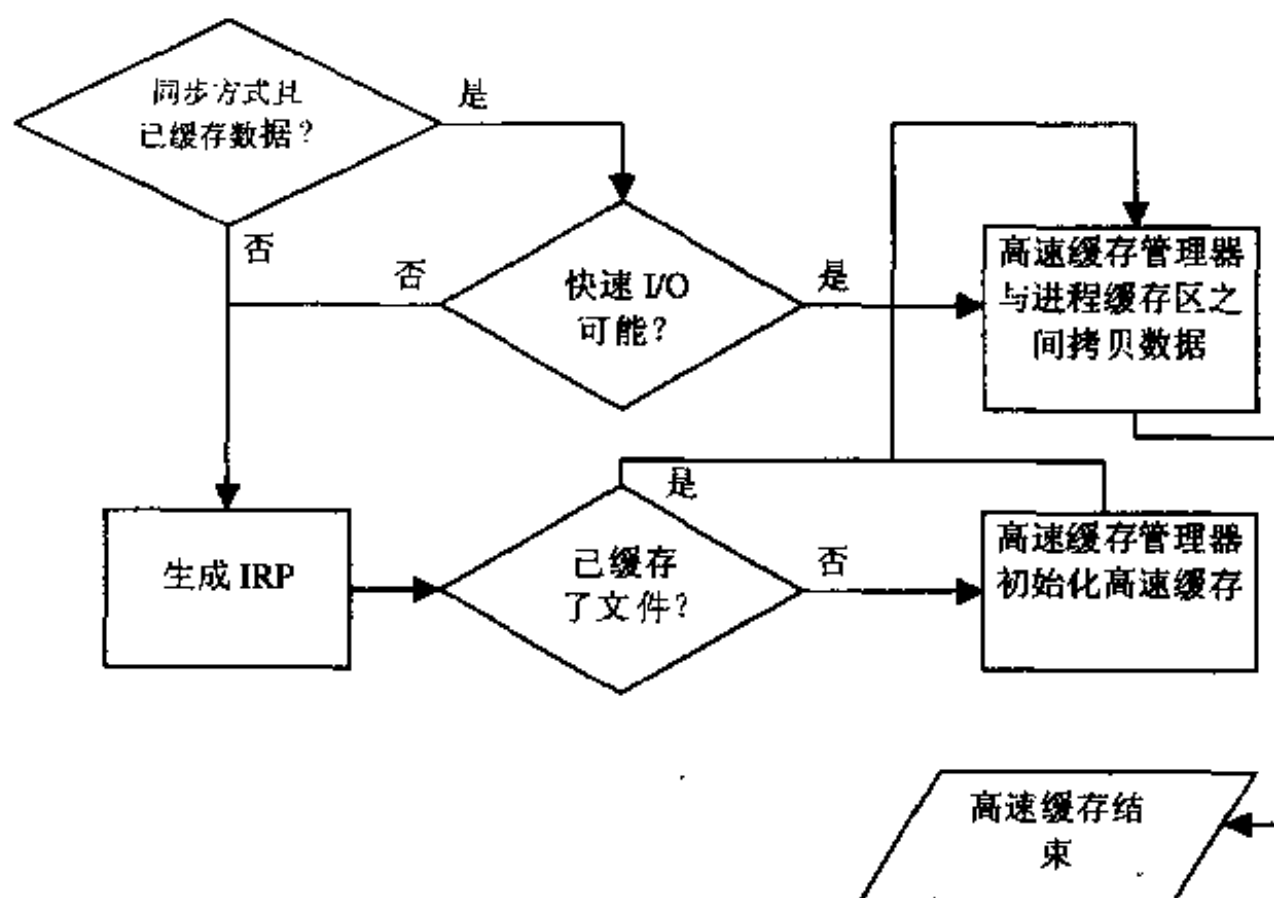


图4-48 快速I/O的决策树

4) 高速缓存管理器将得到的文件偏移转换为高速缓存区的虚拟地址。

5) 对读操作,高速缓存管理器将数据从系统高速缓存中拷贝到请求数据进程的缓冲区中;对写操作,高速缓存管理器将数据从进程的缓存区中拷贝到系统高速缓存。

6) 下面的动作之一发生:

- 对于读,更新调用者私有缓存映射中的预读信息。
- 对于写,设置高速缓存中所有被修改页面的“脏”位,以便延迟写器可以将它写入磁盘。
- 对于通写文件,任何修改被刷新到磁盘。

列在表4-28中的性能计数器或系统变量能够被用来确定系统的快速I/O活动。

表4-28 决定快速I/O活动的系统变量

性能计数器(频度)	系统变量(数值)	描述
Cache: Sync Fast Reads/Sec	CcFastReadWait	快速I/O下的同步读
Cache: Async Fast Reads/Sec	CcFastReadNoWait	快速I/O下的异步读(因为Windows 2000/XP不能异步快速读,所以这总是0)
Cache: Fast Read ResourceMisses/Sec	CcFastReadResourceMiss	因资源冲突而不能进行的快速I/O操作(FAT会出现这种情况,但NTFS不会)
Cache: Fast Read Not Possibles/Sec	CcFastReadNotPossible	不满足条件而没进行的快速I/O操作(文件系统驱动程序决定,如存在字节范围锁定的文件不能使用快速I/O)

4.4.5 高速缓存支持例程

文件的数据第一次被访问时，文件系统驱动程序负责确定文件的某些部分是否被映射到了系统高速缓存。如果没有，文件系统驱动程序必须调用CcInitializeCacheMap函数去设置前面描述过的每个文件的数据结构。

一旦文件设置了高速缓存访问，文件系统驱动程序就可以调用几个函数中的一个来访问文件中的数据。有三种基本的访问缓存数据的方法，每种都适合一种特定的情况：

- (1) “拷贝读取”方法，在系统空间中的高速缓存数据缓冲区和用户空间中的进程数据缓冲区之间拷贝用户数据。
- (2) “映射暂留”方法，使用虚拟地址直接读写高速缓存的数据缓冲区。
- (3) “物理内存访问”方法，使用物理地址直接读写高速缓存的数据缓冲区。

文件系统驱动程序必须提供两种版本的文件读操作——有高速缓存的和没有高速缓存的，以避免内存管理器处理缺页时出现无限循环。当内存管理器通过文件系统从文件中取得数据（当然，通过设备驱动程序）来解决缺页时，它必须在IRP中设置“没有高速缓存”的标志，指出这是一个无需高速缓存的读操作。

下面介绍这三种缓存访问机制、目的和用法。

1. 拷贝读取

因为系统高速缓存位于系统空间，所以它被映射到每一个进程的地址空间。然而，同所有系统空间页面一样，高速缓存的页面不能从用户态被访问，因为那样会造成潜在的安全漏洞。（例如，一个进程可能没有权利读取某些文件在系统高速缓存中的数据。）因此，用户程序读写缓存文件时必须依靠核心态的例程的服务，这些核心态的例程可以在系统空间高速缓存的数据缓冲区和用户进程空间应用程序数据缓冲区之间拷贝数据。文件系统驱动程序可以用来完成这些操作的函数被列在了表4-29中。

表4-29 用于拷贝到缓存和从缓存拷贝的核心态函数

函 数	描 述
CcCopyRead	将指定范围的字节从系统高速缓存中拷贝到用户的缓冲区
CcFastCopyRead	CcCopyRead的更快变种，但限制在32位的文件偏移和同步读（NTFS采用，FAT不用）
CcCopyWrite	将指定范围的字节从用户的缓冲区拷贝到系统高速缓存中
CcFastCopyWrite	CcCopyWrite的更快变种，但限制在32位的文件偏移和同步读（NTFS采用，FAT不用）

2. 映射暂留

正如用户应用程序读写磁盘上的文件一样，文件系统驱动程序需要读写描述文件自身的数据（元数据或卷结构数据）。由于文件系统驱动程序在核心态运行，如果高速缓存管理器得到适当的通知，它们就能够直接修改系统高速缓存中的数据。为了允许这种优化，高速缓存管理提供了表

4-30中列出的一些函数。文件系统驱动程序使用这些函数可以找到文件系统元数据在虚拟内存中的位置，然后可以不借助中间缓冲区而直接修改元数据。

表4-30 用于找到元数据位置的函数

函 数	描 述
CcMapData	映射用于读访问的字节范围
CcPinRead	映射用于读/写访问的字节范围，并且将它暂留内存
CcPreparePinWrite	映射用于写访问的字节范围，并且将它暂留内存
CcPinMappedData	暂留一个以前映射的缓存区
CcSetDirtyPinnedData	告知高速缓存管理器，数据已经被改变了
CcUnpinData	释放内存页，以便它们可以从内存中被除去

如果文件系统驱动程序需要读取在高速缓存中的文件系统元数据，它可以调用高速缓存管理器的映射接口来取得所需数据的虚拟地址。高速缓存管理器找到所有被请求的页面并把它们读入内存，然后将控制权交给文件系统驱动程序。文件系统驱动程序便可以直接地访问这些数据了。

如果文件系统驱动程序需要修改高速缓存页面，它可以调用高速缓存管理器的暂留服务，该服务使被修改的页面驻留内存。那些页面实际上没有被锁在内存中（如同设备驱动程序为进行直接内存访问传输而锁定页面那样）。相反，内存管理器的映射页面写入器看到这些页面是暂留的，于是不把它们写入磁盘，直到文件系统驱动程序释放为止。当页面被释放后，高速缓存管理器将任何变化刷新到磁盘，然后释放元数据占用的高速缓存视图。

映射和暂留接口解决了实现文件系统的一个棘手的问题：缓冲区管理。如果不能直接操纵被缓存的元数据，在更新卷结构时文件系统就必须估计它所需要的缓冲区最大数目。由于高速缓存管理器允许文件系统在缓存中直接访问和修改它的元数据，因此不再需要数据缓冲区，只要简单地更新内存管理器提供的虚拟内存中的卷结构。文件系统受到的唯一限制是可供使用的内存数量。

3. 物理内存访问

除了用于在高速缓存中直接访问元数据的映射和暂留接口外，高速缓存管理器还提供了第三种访问缓存数据的接口：直接存储器存取（direct memory access, DMA）。DMA函数用于不借助缓冲区从高速缓存读取或写入高速缓存，比如网络文件系统在网络上进行传输。

DMA接口将被高速缓存的用户数据的物理地址返回给文件系统（而不是虚地址，虚地址是映射和暂留接口返回的），这个物理地址用于直接从物理内存向网络设备传输数据。虽然少量的数据（1 KB到2 KB）能够用一般的基于缓冲区的接口来传输，但对于大量数据传输，如网络服务器处理远程系统的文件请求，DMA接口能够显著地提高性能。

要描述这些对物理内存的访问，需要使用内存描述链表（MDL）。表4-31中4个独立的函数构成了高速缓存管理器的DMA接口。

表4-31 构成DMA接口的函数

函 数	描 述
CcMdlRead	返回一个描述指定字节范围的MDL
CcMdlReadComplete	释放MDL
CcMdlWrite	返回一个描述指定字节范围的MDL(可能包含0)
CcMdlWriteComplete	释放MDL, 将那个字节区范围记上“写”

4.4.6 写阻塞

Windows 2000/XP 必须确认调度写操作是否会影响系统的性能,然后再安排各项延迟写操作。首先,它询问现在立刻写入一定数量的字节是否会损害性能,如果必要阻塞该项写操作。接下来,它设置当写操作再次被允许时自动写入字节的回调。一旦获悉将要进行的写操作,高速缓存管理器便会判断高速缓存中有多少脏页和有多少可以使用的物理内存。如果空闲的物理内存页不足,高速缓存管理器立即阻塞请求向高速缓存中写数据的文件系统线程。高速缓存管理器的延迟写器会将一些脏页刷新到磁盘,然后允许被阻塞的文件系统线程继续。当文件系统或网络服务器进行大量写操作时,这种写阻塞机制防止了系统的性能由于缺少内存而下降。

写阻塞对于网络重定向程序在低速传输的线路上传送数据也很有用。例如,假设一个本地进程通过9600波特率的线路向远程文件系统写大量数据。这些数据直到高速缓存管理器的延迟写器刷新高速缓存时才被写入远程的磁盘。如果重定向程序积累了大量刷新到磁盘的脏页,那么接收者在数据传输结束前可能会接收到一个网络超时。通过使用CcSetDirtyPageThreshold函数,高速缓存管理器允许网络重定向程序设置一个可以接受的高速缓存脏页数目界限,以防上述情况的发生。通过限制脏页的数量,重定向程序保证了高速缓存刷新操作不会引起网络超时。

4.5 小结

Windows 2000/XP 高速缓存管理器提供了一种高速、智能的机制,用以减少磁盘I/O和增加系统的整体吞吐量。基于虚拟块的高速缓存使Windows 2000/XP 高速缓存管理器能够进行智能预读。依靠全局内存管理器的映射文件机制访问文件数据,高速缓存管理器提供了特殊的快速I/O机制减少了用于读写操作的时间,而且将与物理内存有关的管理工作交给了Windows 2000/XP全局内存管理器,这样减少了代码的冗余,提高了效率。

习题

- 4.1 试述虚拟存储器的定义及其特征。
- 4.2 某计算机有四个页框。每一页装入的时间、最后一次访问的时间以及访问位和修改位如下所示(时间用时钟的点数来表示):

page loaded last ref R M

0	126	279	0	0
1	230	260	1	0
2	120	272	1	1
3	160	280	1	1

- a) FIFO算法将置换哪一页?
 - b) LRU算法将置换哪一页?
 - c) CLOCK(NRU)算法将置换哪一页?
- 4.3 在一个请求页式存储系统中，一个程序的页面走向为4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 1, 5, 并采用LRU页面置换算法。假设分配给该程序的存储块数为3和4时，给出访问过程中发生的缺页次数和缺率。
- 4.4 论述Windows 2000如何利用Intel x86结构对页式存储管理的支持，实现页式虚拟存储器。
- 4.5 结合图4-27说明在Windows 2000中，物理页大致分为几种状态，以及这些状态之间是如何转换的。
- 4.6 Windows 2000的高速缓存系统有那些特点?
- 4.7 在Windows 2000的高速缓存管理器中，采用了多级VACB数组。这种优化是针对哪种情况的？有何优越性？
- 4.8 为了避免当内存管理器遇到缺页时出现无限循环，高速缓存管理器对文件系统驱动程序有何要求？
- 4.9 分析Windows 2000对其多分区卷，如条带卷、镜像卷、RAID_5卷，采取了哪些优化措施？这些优化措施对性能提高有何帮助？
- 4.10 Windows 2000为什么可以和其他类型的操作系统（如Linux）共存于一块硬盘之上？在共存的情况下，系统是如何实现双引导的？

第 5 章

文件系统

第 ⑤ 章

文件系统

计算机的主要功能之一就是对数据进行数值或非数值计算。操作系统作为计算机的最为重要的系统软件必须提供数据存储、数据处理、数据管理的基本功能。数据存储通常是以文件形式存放在磁盘或其他外部存储介质上，数据处理是通过文件处理来进行的，数据管理是通过文件管理来完成的。文件管理是通过目录来完成的，而目录又是建立在分区或卷的基础上的。操作系统中与文件和目录相关的子系统称为文件系统。文件系统在操作系统中占有非常重要的地位。

本章将首先介绍文件系统的有关概念，接着分析文件系统性能要求和设计原理，然后以NTFS文件系统为例来具体讨论文件系统的实现。

5.1 文件概念与实现

5.1.1 文件

文件是具有一定名称的一组相关数据的集合。文件通常存储在外部存储介质上（如磁盘、光盘等）。下面我们从文件命名、文件分类、文件属性、文件存取、文件结构和文件操作等方面，讨论文件的有关概念。

1. 文件命名

文件提供了一种将数据保存在外部存储介质上以便于访问的功能。为了方便用户使用，每个文件都有特定的名称。这样用户就不必关心文件存储方法、物理位置以及访问方式等，而可以直接通过文件名来使用文件。

各种文件系统的文件命名不尽相同。文件名称的长度因系统而异。例如：有的文件系统如FAT12（一种老的MS-DOS所用的文件系统）的8.3命名规则规定文件名为8个字符，外加句点和3个字符的扩展名；有的如NTFS（New technology file system）则可以达到255个字符；而有的如ext2（一种Linux文件系统）则没有长度限制。

有的文件系统不区分文件名的大小写，而有的则加以区分。FAT12属于前者，ext2则属于后者。例如，chap5.htm，CHAP5.HTM、Chap5.htm和Chap5.Htm等文件名称在FAT12中表示同一文件，而在ext2中则表示不同的文件。

有的文件系统只能使用ASCII字符命名文件，而有的则可以使用更为广泛的字符如Unicode。例如FAT12属于前者，而NTFS属于后者。

有的操作系统对不同的后缀有特定的解释，而有的则没有统一的规定。MS-DOS和Windows 2000/XP属于前者，而UNIX属于后者。例如，在MS-DOS中，`prog.c`为C语言源文件，`prog.cxx`为C++源文件，`prog.doc`为Word文件，`prog.hlp`为帮助文件，`prog.htm`为HTML文档，`prog.ini`为配置文件等。

2. 文件属性

文件可包括两个部分内容：一是文件所包含的数据，常称为文件数据；二是关于文件本身的说明信息或属性信息，常称为文件属性。文件属性主要描述文件的元信息，如创建日期、文件长度、文件权限等，这些信息主要被文件系统用来管理文件。不同的文件系统通常有不同种类和数量的文件属性。下面简要讨论一些常用的文件属性：

- 文件名称：文件名称是供用户使用的外部标识。这是文件最基本的属性。每个文件都必须有个名称以用于标识。文件名称通常由一串ASCII码或者汉字构成，现在常常由Unicode组成。
- 文件内部标识：有的文件系统不但为每个文件规定了一个外部标识，而且规定了一个内部标识。文件内部标识只是一个编号，可以方便管理和查找文件。在UNIX文件系统中，`i-node`就是内部标识。
- 文件物理位置：具体标明文件在存储介质上所存放的物理位置。例如，对于按连续区域分配的文件，需要给出起始的物理块号和文件长度；对于按索引方式组织的文件，需要给出索引表所在物理块号和索引长度。
- 文件拥有者：操作系统通常为多用户的，不同的用户也拥有各自不同的文件，对这些文件的操作权限也不同。通常文件创建者对自己所建的文件拥有一切权限，而对其他用户所建的文件则拥有有限的权限。为了更好地管理各个用户，需要为多用户操作系统所使用的文件加上文件拥有者的属性。
- 文件权限：通过文件权限，文件拥有者可以为自己的文件赋予各种权限，如可允许自己读写和执行，允许同组的用户读写，而只允许其他用户读。
- 文件类型：可以从不同的角度来对文件进行分类，例如普通文件或是设备文件，可执行文件或不可执行文件，等等。
- 文件长度：文件长度通常是其数据的长度，也可以是允许的最大长度。长度单位通常是字节，也可以是块。
- 文件时间：文件时间有很多，如最初创建时间，最后一次的修改时间，最后一次的执行时间，最后一次的读时间等。

3. 文件分类

为了有效、方便地组织和管理文件，常按照某种观点对文件进行分类。文件分类方法有很多，这里简要介绍几种常用的分类方法。

按文件的用途进行分类：

- 系统文件：包括操作系统内核、系统应用程序等。这些通常都是可执行的二进制文件，但有的也可能是文本文件，如配置文件等。这些文件对于系统的正常运行是必不可少的。

- 库文件：这包括标准的和非标准的子程序库。标准的子程序库通常称为系统库，提供对系统内核的直接访问，而非标准的子程序库则是提供满足特定应用的库。库文件又分为两大类：一类是动态链接库，另一类是静态链接库。
- 用户文件：用户自己的文件，如用户的源程序、可执行程序 and 文档等。

按文件的性质进行分类：

- 普通文件：主要是系统所规定的普通格式的文件，例如字符流组成的文件，它包括用户文件、库函数文件、应用程序文件，等等。
- 目录文件：包含普通文件与目录的属性信息的特殊文件，这主要是为了更好地管理普通文件与目录。
- 特殊文件：在UNIX系统中，所有的输入输出设备都被看作是特殊的文件，甚至在使用形式上也和普通文件相同。通过对特殊文件的操作可完成相应设备的操作。

按文件的保护级别进行分类：

- 只读文件：允许授权用户读，但不能写。
- 读写文件：允许授权用户读写。
- 可执行文件：允许授权用户执行，但不能读写。
- 不保护文件：所有用户都有一切权限。

按文件数据的形式进行分类：

- 源文件：源代码和数据构成的文件。
- 目标文件：指的是源程序经过编译程序编译，但尚未链接成可执行代码的目标代码文件。
- 可执行文件：编译后的目标代码由连接程序连接后形成的可以运行的文件。

除了以上的分类方法外，还可以按照文件的其他属性进行分类。由于各种系统对文件的管理方式不同，因而对文件的分类方法也有很大的差异，但是其根本目的都是为了提高文件的处理速度以及更好地实现文件的保护和共享。

4. 文件存取

文件存取方式是指用户在使用文件时按何种次序存取文件。文件存取方式主要有顺序访问、随机访问、索引访问等，下面作一简要介绍。

文件顺序访问是按从前到后的顺序对文件进行读写操作。这种存取方式最为简单。有的存储设备如磁带只能支持顺序访问。

文件随机访问，也称为直接访问，可以按任意的次序对文件进行读写操作。有的存储设备如磁盘能支持随机访问（当然也能支持顺序访问）。

文件索引访问，也称按键访问，这种方式对文件中的记录按某个数据项（通常称为键）的值来排列，从而可以根据键值来快速存取。如索引表很长，则可以将索引表再加以索引，以形成具有层次结构的多级索引。如果将记录块的物理位置作为键值，那么可以将随机访问作为索引访问的特例。

5. 文件操作

为了方便用户使用文件系统，文件系统通常向用户提供各种调用接口。用户通过这些接口来

对文件进行各种操作。对文件的操作可以分为两大类：一类是对文件自身的操作，例如，建立新文件，打开文件，关闭文件，读写文件，等等；另一类是对记录的操作（最简单的记录可以是一个字符），例如，查找文件中的字符串，以及插入和删除，等等。以下是一些常用的文件操作。

- 文件创建：创建文件时，系统会进行各项子操作。首先，系统会为新文件分配所需的外存空间，并且在文件系统的相应目录中，建立一个目录项，该目录项记录了新文件的文件名及其在外存中的地址等文件属性。
- 文件删除：当已经不再需要某个文件时，便可以把它从文件系统中删除。这时执行的是和创建新文件相反的操作。系统先从目录中找到要删除的文件项，使之成为空项，紧接着回收该文件的存储空间，用于下次分配。
- 文件截断：如果一个文件的内容已经很陈旧而需要进行全部更新时，虽然我们可以先删除文件再建立一个新文件，但是如果文件名及其属性并没有发生变化时，可截断文件。即将原有文件的长度设为0，也可以说是放弃文件的内容。
- 文件读：通过读指针，将位于外部存储介质上的数据读入到内存缓冲区。
- 文件写：通过写指针，将内存缓冲区中的数据写入到位于外部存储介质上的文件中。
- 文件的读写定位：前面介绍的读写操作只是提供了文件的顺序存取手段，而若对文件的读写进行定位操作，也即改变读写指针的位置，则可以从文件的任意位置开始读写，为文件提供随机存取的能力。
- 文件打开：在开始使用文件时，首先必须打开文件。这可以将文件属性信息装入内存，以便以后快速查用。
- 文件关闭：在完成文件使用后，应该关闭文件。这不但是为了释放内存空间，而且也因为许多系统常常限制可以同时打开的文件数。

6. 文件结构

文件结构是指文件的组织形式。文件结构分为文件的逻辑结构（file logical structure）和文件的物理结构（file physical structure）。前者是从用户的观点出发，所看到的是独立于文件物理特性的文件组织形式，是用户可以直接处理的数据及其结构。而后者则是文件在外存上具体的存储结构。

文件的逻辑结构对用户而言是透明的，以方便用户存取。文件的逻辑结构较简单，一般可分为记录式文件和流式文件两种。前者是指用户把每个文件分为若干记录单位，存取文件是以记录为单位来进行的，而后者则是指文件由字符流组成，文件内部的信息不再划分单位。

文件的物理结构则是指文件在外部存储介质上如何存放，也叫文件的存储结构。它对文件的存取方法有较大的影响。关于文件的物理结构，将在文件实现一节中作详细介绍。

5.1.2 文件实现

文件实现的主要问题是如何在外部存储介质上为创建文件而分配空间，为删除文件而回收空间，以及对空闲空间进行管理。磁盘可以随机存取的特性非常适合文件系统的实现，因此磁盘是最常用的文件系统实现的外部存储介质。这里主要讨论两个问题：一是磁盘空闲空间的分配，二是磁盘空闲空间的有效管理。

1. 空间分配策略

在大多数情况下，许多文件是存放在同一个磁盘上的。此时，主要问题是如何为这些文件分配空间，使得在有效利用磁盘空间的同时，可以快速存取文件。常用的磁盘空间分配策略主要包括连续空间分配、链接空间分配、索引空间分配等。每种策略都有各自的优缺点。一般情况下，一个文件系统只采用一种策略来分配文件空间。

(1) 连续空间分配

连续空间分配是最简单的磁盘空间分配策略，参见图5-1。每一个文件都占据了一个完整且连续的磁盘区域。对于这样的文件，由于空间的连续性，当访问下一个磁盘块时，通常无需移动磁头，而只有当磁头从一个磁道的最后一个块移向下一个磁道的第一个块时，才需要移动磁头。因此这种分配策略的磁头移动次数最少。对于这类文件，目录通常只需包括文件名、文件块的起始地址和文件长度。

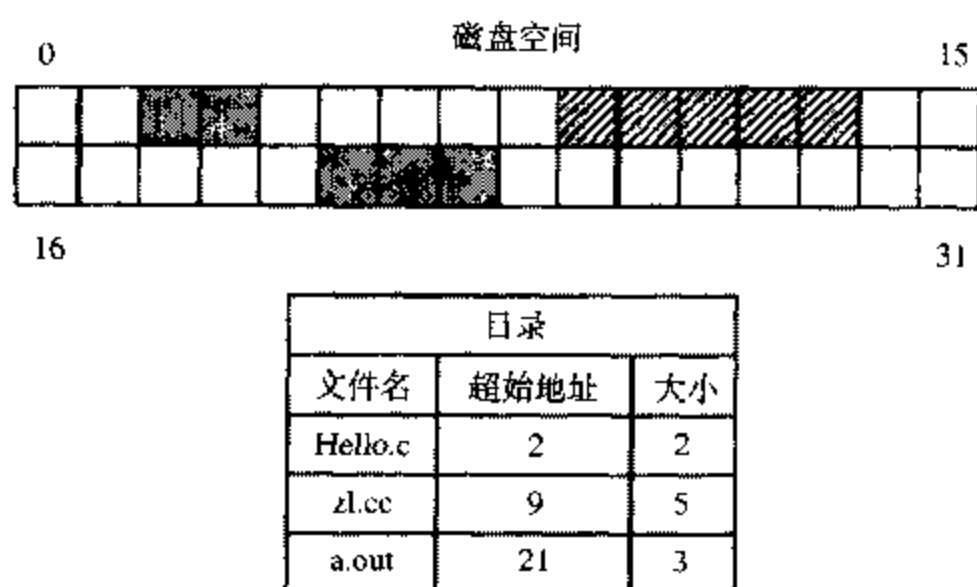


图5-1 连续空间分配

这种分配策略的优点是实现简单，存取速度快。只要该文件是连续存放的，第一个块号和块数就可以确定该文件在外部存储介质上的位置。缺点是文件长度不易动态增加，因为一个文件的末尾处的空块可能已经分配给别的文件，一旦需要增加，就需要大量的改动。另外，反复增删以后，存储设备中便会产生类似与内存分配中出现的磁盘空间碎片，因而只适用于长度固定的文件。

(2) 链接空间分配

对于链接空间分配，参见图5-2。每一个文件都有一张相应的磁盘块的链表。这些磁盘块可以分散在磁盘的任何地方，除了最后一个磁盘块外，每一个磁盘块都有一个指针指向下一个磁盘块。这些指针对用户是透明的。对于采用链接空间分配的文件，目录项通常只需包括文件名、文件的开始块和结束块。

这种分配策略的优点是没有外部碎片，每一个空闲块都可以用来分配。并且只要有空闲块的存在，一个文件就可以任意地增长而没有其他的限制。缺点是只有在顺序访问时，链接空间分配策略才是高效的。为了访问文件的第*i*块，必须从第一块开始访问，然后根据指针访问下一块，直到找到第*i*块。每一次都需要读写磁盘，有时还需要移动磁头来寻道。另一个缺点是必须给指针分配空间。

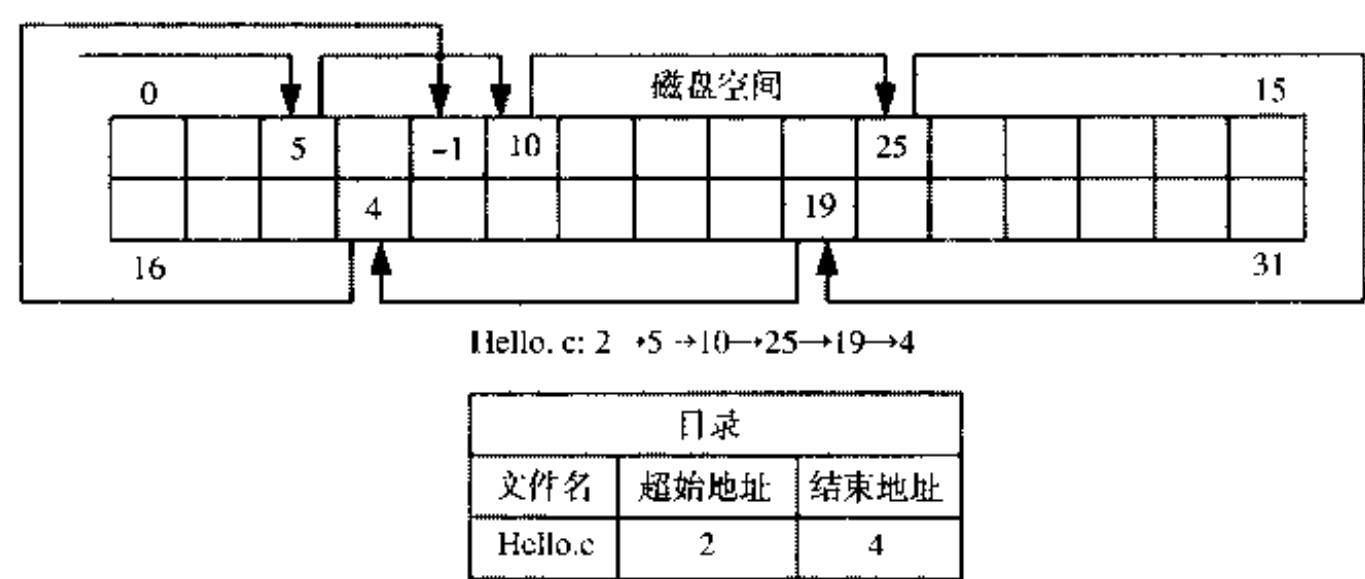


图5-2 链接空间分配

(3) 索引空间分配

对于索引空间分配策略，参见图5-3。每一个文件都有一个索引块。这个索引块就是一个表，每一个表项存放文件所占有的单个磁盘块的地址。表的第*i*项就是指向文件的第*i*个磁盘块。对于这类文件，目录的每一项可包括文件名和文件索引块的地址。

这种分配策略不但避免了连续空间分配存在的外部碎片问题和文件长度受限制的问题，而且还支持对任何一个文件块的直接访问。其缺点是由于索引块的分配增加了系统存储空间的开销。对于索引空间分配策略，索引块的大小选择是一个很重要的问题。为了节约磁盘空间，希望索引块越小越好。但索引块太小无法支持大文件。所以要采用一些技术来解决这个问题，例如可以采用多级索引技术。

另外，存取文件需要两次访问外存——首先要读取索引块的内容，然后再访问具体的磁盘块，因而降低了文件的存取速度。为了克服这个缺点，通常在读取文件之前，先将磁盘上的索引块读入并保存在内存缓冲区中，以便加快文件访问速度。

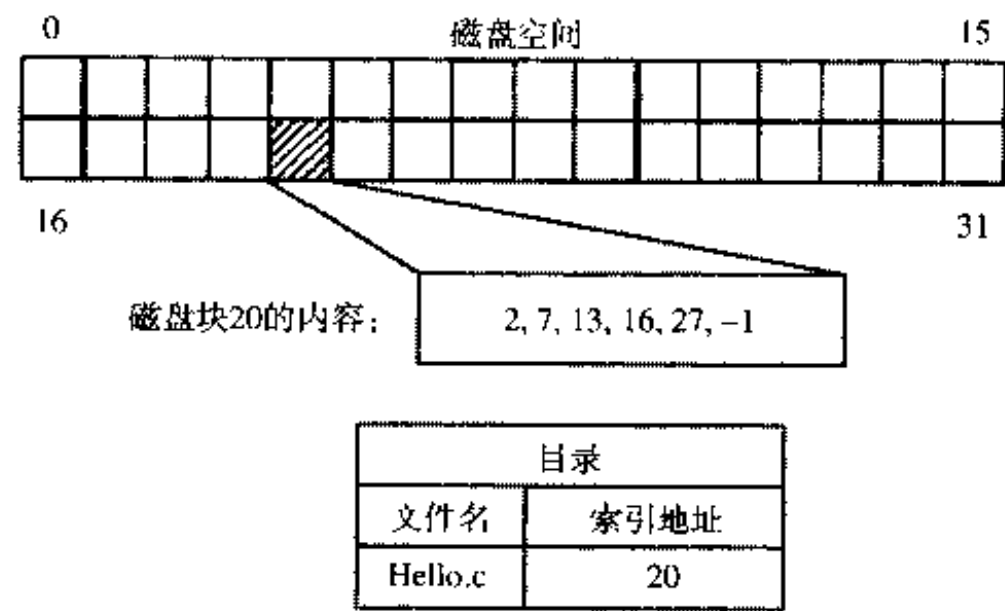


图5-3 索引空间分配

(4) 组合空间分配

组合空间分配是多种分配策略的组合，参见图5-4。例如，在UNIX文件系统中，每个文件都

有一个对应的I节点（I-node），其中可以有15个指针用于空间分配。前12个指针指向可以直接访问的磁盘块。这样对一般的小文件来说，一个索引块就足够用了。剩下的3个指针指向间接块，即不包含文件数据的块。第一个指针指向一级间接块。一级间接块也是索引块，包含的不是数据，而是指向数据块的指针。类似地，第二个指针指向的二级间接块，其中包含指向一级间接块的指针。第三个指针指向的是三级间接块。对于这类文件，目录项可包括文件名和I节点。

这种分配策略的优点是不但具有以上多种分配策略的所有优点，而且非常灵活。这种分配策略已经在众多UNIX文件系统中得到广泛使用。

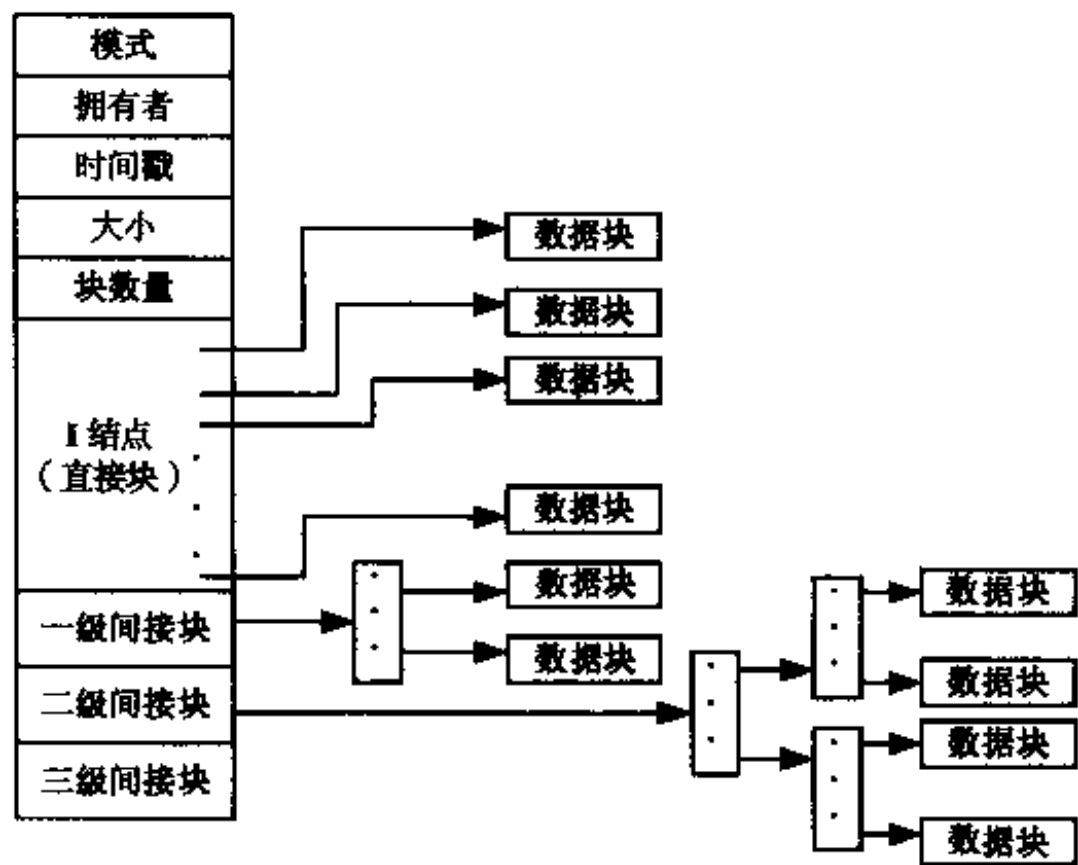


图5-4 组合空间分配

2. 空闲空间管理

因为磁盘空间的容量总是有限的，重新把删除文件后得到的空闲空间分配给新的文件就很必要。这就要求文件系统随时掌握磁盘空闲空间的情况，以便随时分配给新的文件或目录。为了记录空闲磁盘空间，系统通常维持一个空闲空间表。这个表记录了所有的空闲块，也就是那些尚未分配给文件或目录的块。

空闲空间表的实现方法有很多种，下面将讨论几种常用的实现方法。

(1) 空闲块位示图

空闲空间表可由位图或位矢量的方法来实现。每一个磁盘块由1位（bit）来表示。如果该磁盘块是空闲的，这个位就置1；否则，就置0。例如，图5-5显示了一个磁盘，其中第2、4、5、10、11、12、13、14、15、22、23、24、25、26、27、28等块为空闲的，其他的块为已分配的。

00101100001111110000001111111000

图5-5 空闲块位图

这个方法的主要优点是简单，寻找空闲块或寻找n个连续的空闲块很有效。实际上有很多处理器支持一些位操作指令，这些指令可用来有效地实现

上述操作。缺点是除非整个位图都装载入内存中，否则这个方法的效率不高。而对大磁盘来说，这个方法是很耗费内存的，因而不常用。

(2) 空闲块链表

空闲块链表是通过链表来实现的，它把磁盘上的所有的空闲块链接在一起。当系统需要给文件分配存储空间时，分配程序从空闲块链表的链首摘取所需的若干块，链首指针相应后移。与此相反，当删除文件回收空闲块时，则把释放的空闲块添加到空闲块链的链尾上。空闲块链表的实现方法因系统而异。有的按照空闲区大小进行连续链接，有的按照释放先后进行链接，最常用的是成组链接法。图5-6表示了图5-5中的磁盘空闲空间用块链表形式表示的情况。

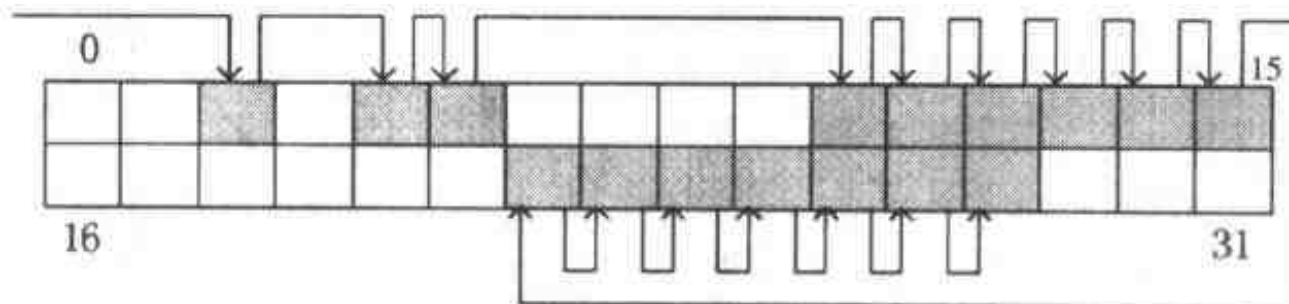


图5-6 空闲块链接

这个方法的优点是内存消耗少，缺点是空闲空间表遍历的效率低。

(3) 其他空闲块管理方法

可以对空闲块链表作一简单改变。由于大量的空闲块通常是连续的，所以可以将空闲空间用大小不一的连续块（而不是一块接一块）来链接，其中每一个链节点由连续块的起始地址和块数的组合来表示。这样不但提高空间分配的速度，而且也常常降低了存储消耗。

除了以上方法外，还有许多其他空闲块管理方法。限于篇幅，这里就不多作讨论了。

5.2 目录概念与实现

建立文件系统的主要任务之一，是为了让用户借助于文件系统可以很方便地访问外存。在现代计算机系统中，通常都要存储大量的文件，为了有效地管理这些文件，必须对它们加以适当组织。这可以通过目录来实现。本节将从用户接口角度和实现角度来对目录进行讨论。

5.2.1 目录

1. 目录功能

目录最基本的功能就是通过文件名可以快速方便地获取文件的属性信息，如文件物理位置等。一般来说，目录应具有如下几个功能：

- 实现“按名操作”。用户只需提供文件名，就可以对文件进行操作，这既是目录管理的最基本功能，也是文件系统向用户提供的最基本服务。
- 提高检索速度。这就需要在设计文件系统时合理地设计目录结构。对于大型系统来说，这是一个很重要的设计目标。
- 允许文件同名。为了便于用户按照自己的习惯来命名和使用文件，文件系统应该允许对不

同文件使用相同名称。这时，文件系统可以通过不同工作目录来加以解决。

- 允许文件共享。在多用户系统中，应该允许多个用户共享一个文件，这样就可以节省文件的存储空间，也可以方便用户共享文件资源。当然，还需要相应的安全措施，以保证不同权限的用户只能取得相应的文件操作权限，防止越权行为。

2. 目录结构

这里主要从逻辑角度即用户角度来讨论目录结构。根据目录的结构，可以将目录分为：单级目录、二级目录、多级层次目录、无环图结构目录、图状结构目录等。下面，分别作一简要介绍。

(1) 单级目录

单级目录最为简单（参见图5-7），在整个文件系统中仅仅建立和维护一张总的目录表，系统上的所有文件都在该表中占有一项。当存取文件时，用户只要给出文件名，系统通过查找这个目录表，找到文件名相对应的项就可获得该文件的属性信息。在通过访问权限验证后，就可以根据目录项中提供的文件物理地址对文件实施存取操作。在建立文件时，只要在目录表中申请一个空闲项，并填入文件名及其相关属性信息即可。同样，在删除文件时，只要把相应的目录项标记为空闲项，并回收空间即可。

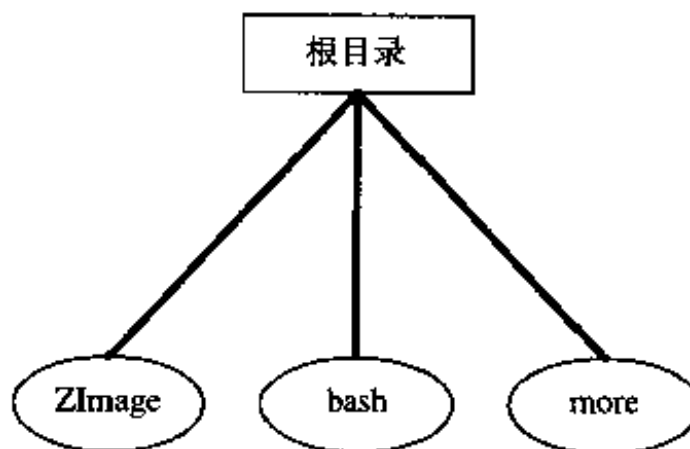


图5-7 单级目录

这种单级目录的主要优点是实现简单，但是存在明显的缺点：

- 不允许文件重名。单级目录下的文件，不允许和另一个文件有相同的名字。但是对于多用户系统来说，这又是很难避免的。即使是单用户环境，当文件数量很大时，也很难弄清到底有哪些文件，这就导致文件系统极难管理。
- 文件查找速度慢。对于稍具规模的文件系统来说，由于拥有大量的目录项，致使查找一个指定的目录项可能花费较长的时间。

(2) 二级目录

二级目录可以解决文件重名，即把系统中的目录分为一个主目录表（Master File Directory, MFD）和多个次目录表（User File Directory, UFD），参见图5-8。在多用户系统中，一般每个用户都拥有一个属于自己的次目录表UFD，而主目录表MFD则存储着各个UFD的信息，说明各个UFD的名称、物理位置等。

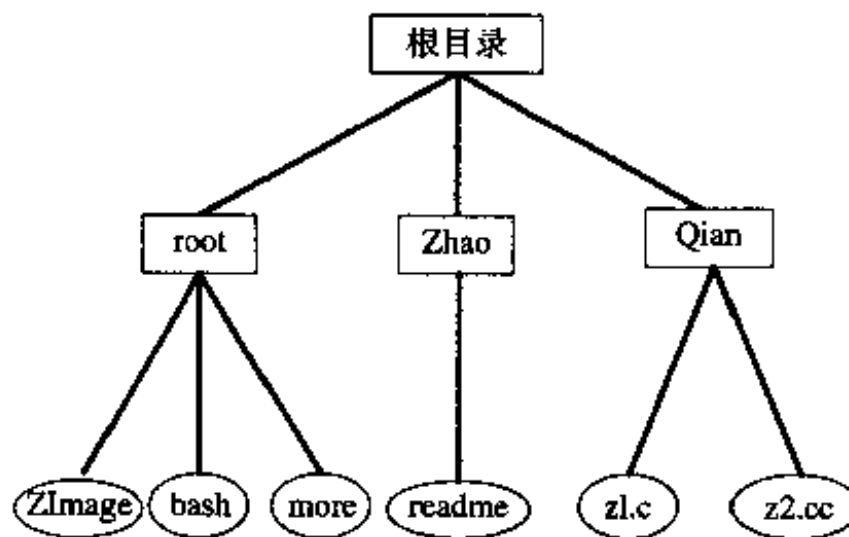


图5-8 二级目录

当使用文件时，用户必须给出用户名和文件名。系统根据用户名在主目录中找到该用户目录，再根据文件名在用户目录中找到文件的物理地址。因此，即使不同的用户给文件取了相同的名字，也不会造成混乱。当增加新用户时，系统为他建立一个用户目录。当删除用户时，则可撤消其目录。

这种目录结构基本上克服了单级目录的缺点，并具有以下优点：

- 提高了文件检索速度。这是因为指明了用户名大大缩小了需要检索的文件数量。
- 部分地允许文件名重名。在不同的目录中，可以使用同一个文件名。这是因为它们被不同的目录空间所分隔。

二级目录也有其缺点，如对同一用户，也不能有两个同名的文件存在。

(3) 多级层次目录

多级层次目录（也叫树结构目录）是二级目录的推广，参见图5-9。为了更好地反映系统中众多文件的不同用途，也为了方便查找文件，可以把二级目录加以推广，而形成多级层次目录。在多级层次目录中，有一个主目录和许多分目录。分目录不但可以包含文件，而且还可以包含下一级的分目录。这样依次推广下去就形成了多级层次目录。

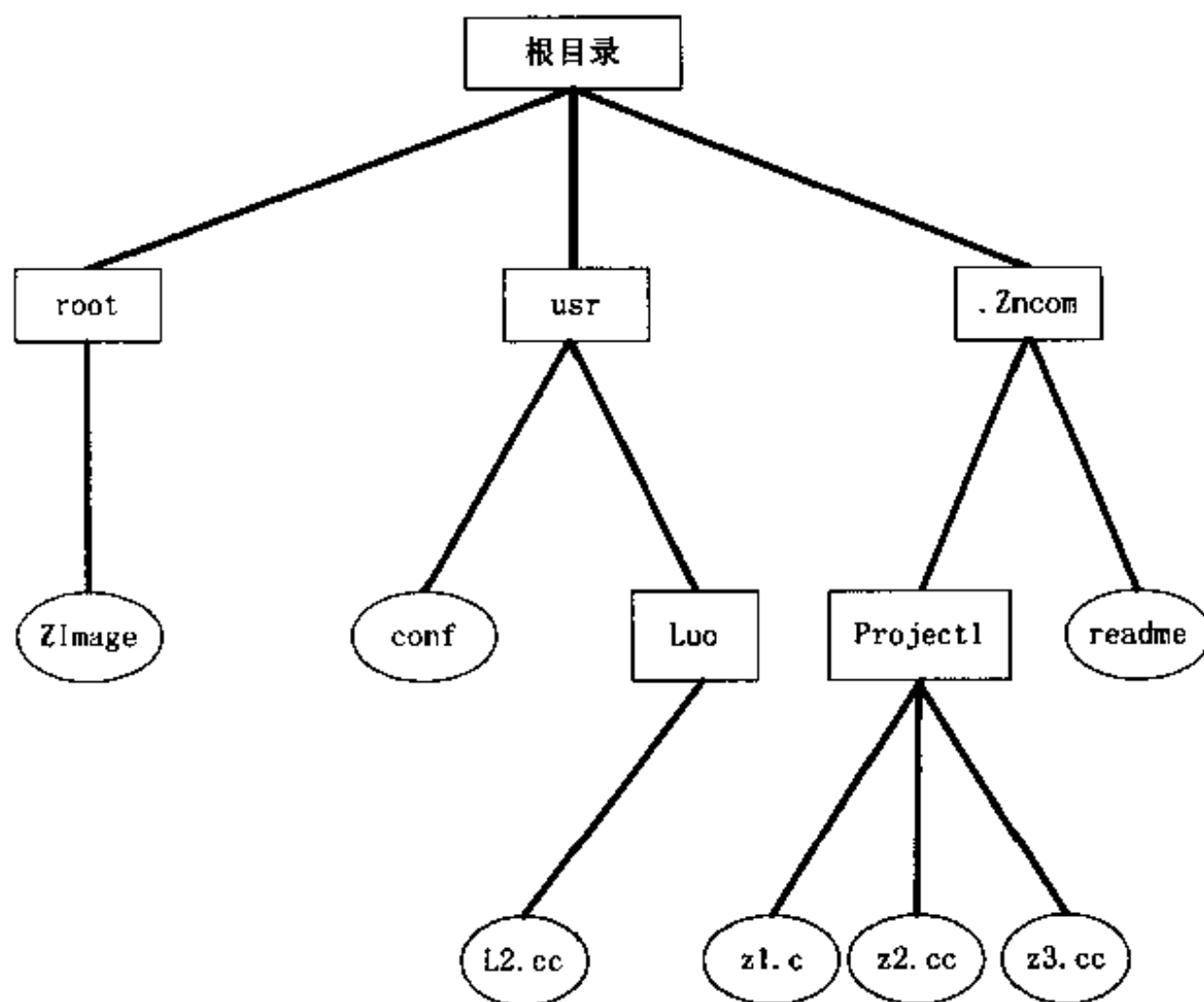


图5-9 多级目录

多级目录具有以下优点：

- 既可方便用户查找文件，又可以把不同类型和不同用途的文件分类。
- 允许文件重名。不但不同用户可以使用相同名称的文件，同一用户也可使用相同名称的文件。
- 利用多级层次结构关系，可以更方便地制定保护文件的存取权限，有利于文件的保护。

多级层次目录也有其缺点，如不能直接支持文件或目录的共享。

(4) 无环结构目录

无环结构目录是多级目录的推广。多级层次目录不直接支持文件或目录的共享。为了允许文

部存储介质中。目录创建也就是在外部存储介质中，创建一个目录文件以备存取文件属性信息。

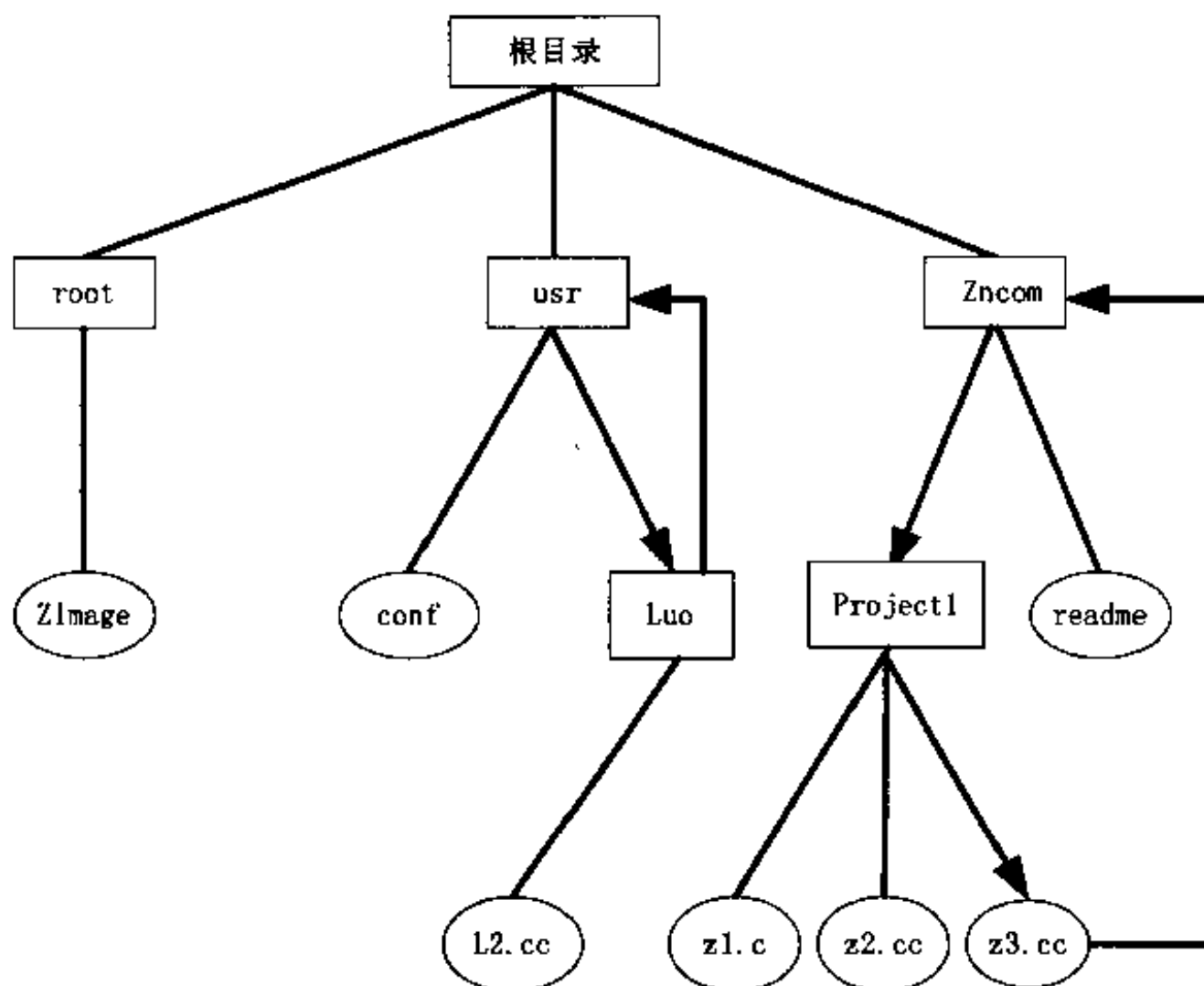


图5-11 图状结构目录

- 目录删除：也就是从外部存储介质中，删除一个目录文件。通常而言，只有当目录为空时，才能删除，有的系统中，删除一个非空的目录，也常常删除其中的所有文件与子目录。
- 文件检索：要实现用户对文件的按名存取，这就涉及到文件目录的检索。系统按下面的步骤为用户找到所需的文件：首先，系统利用用户提供的文件名，对文件目录进行查询，以找到相应的属性信息；然后，根据这些属性信息，得出文件所在外部存储介质的物理位置；最后，如需要可启动磁盘驱动程序，将所需的文件数据读到内存中。
- 目录打开：如要用的目录不在内存中，应从外存中打开并读入相应的目录文件。
- 目录关闭：当所用目录使用结束后，应关闭目录以释放内存空间。

5.2.2 目录实现

目录实现的算法对整个文件系统的效率、性能和可靠性有很大的影响。下面对这些算法作一简单讨论。

1. 线性表算法

目录实现的最简单的算法是一个线性表，每个表项由文件名和指向数据块的指针组成。当要搜索一个目录项时，可采用线性搜索。这个算法实现简单，但运行很耗时。比如创建一个新

的文件时，需要先搜索目录以确定没有同名文件存在，然后再在线性表的末尾添加一条新的目录项。

线性表算法的主要缺点就是寻找一个文件时要做线性搜索。目录信息是经常使用的，访问速度的快慢会被用户觉察到。所以很多操作系统常常将目录信息放在高速缓存中。对高速缓存中的目录的访问可以避免磁盘操作，以加快访问速度。当然也可以采用有序的线性表，使用二分搜索来降低平均搜索时间。然而，这会使实现复杂化，而且在创建和删除文件时，必须始终维护表的有序性。

2. 哈希表算法

采用哈希表算法时，目录项信息存储在一个哈希表中。进行目录搜索时，首先根据文件名来计算一个哈希值，然后得到一个指向表中文件的指针。这样该算法就可以大幅度地减少目录搜索时间。插入和删除目录项都很直观，只需要考虑一下两个目录项冲突的情况，就是两个文件名返回的数值一样的情形。哈希表的主要难点是选择合适的哈希表长度与适当的哈希函数。

3. 其他算法

除了以上方法外，还可以采用其他数据结构，如B+树。NTFS文件系统就使用了B+树来存储大目录的索引信息。B+树数据结构是一种平衡树。对于存储在磁盘上的数据来说，平衡树是一种理想的分类组成方式，这是因为它可以使得查找一个数据项所需的磁盘访问次数减到最小。

由于使用B+树存储文件，文件按顺序排列，所以可以快速查找目录，并且可以快速返回已经排好序的文件名。同时，因为B+树是向宽度扩展而不是深度扩展，NTFS的快速查找时间不会随着目录的增大而增加。

5.3 文件系统

以上从接口和实现角度讨论了文件系统文件与目录这两个重要概念及其实现方法。本节将从整体角度来讨论文件系统。这包括三个方面的内容：文件系统模型、文件系统可恢复性和文件系统安全性。

5.3.1 文件系统模型

1. 文件系统的层次模型

文件系统的传统模型为层次模型，该模型由许多不同的层组成。每一层都会使用下一层的功能特性来创建新的功能，为上一层服务。每一层都在下层的基础上，向上层提供更多的功能，由下至上逐层扩展，从而形成一个功能完备，层次清晰的文件系统。参见图5-12。

层次模型的分层方法有很多。这里介绍一下常用的四层模型。该模型包括基本I/O控制层、基本文件系统层、文件组织模块层和逻辑文件系统层：

- 基本I/O控制层：由设备驱动程序和中断处理程序组成，

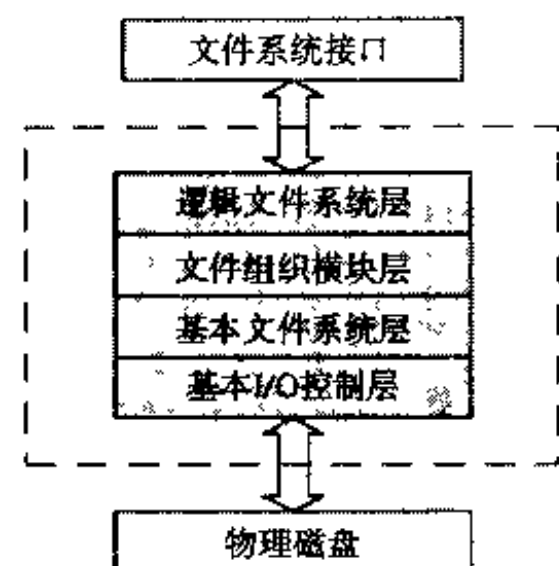


图5-12 文件系统模型

实现内存和磁盘系统之间的信息传输。

- 基本文件系统层：主要向相应的设备驱动程序发出读写磁盘物理块的一般命令。
- 文件组织模块层：负责对具体文件以及这些文件的逻辑块和物理块进行操作。
- 逻辑文件系统层：使用目录结构为文件组织模块提供所需的信息，并负责文件的保护和安全。

2. SUN虚拟文件系统模型

以上简单的文件系统层次模型对支持单个文件系统比较合适，而对于同时支持多个文件系统则有所不足。为此，SUN公司提出虚拟文件系统（Virtual File System, VFS）框架结构，参见图5-13。通过VFS可以支持多种文件系统，如ext2，fat，ntfs等。

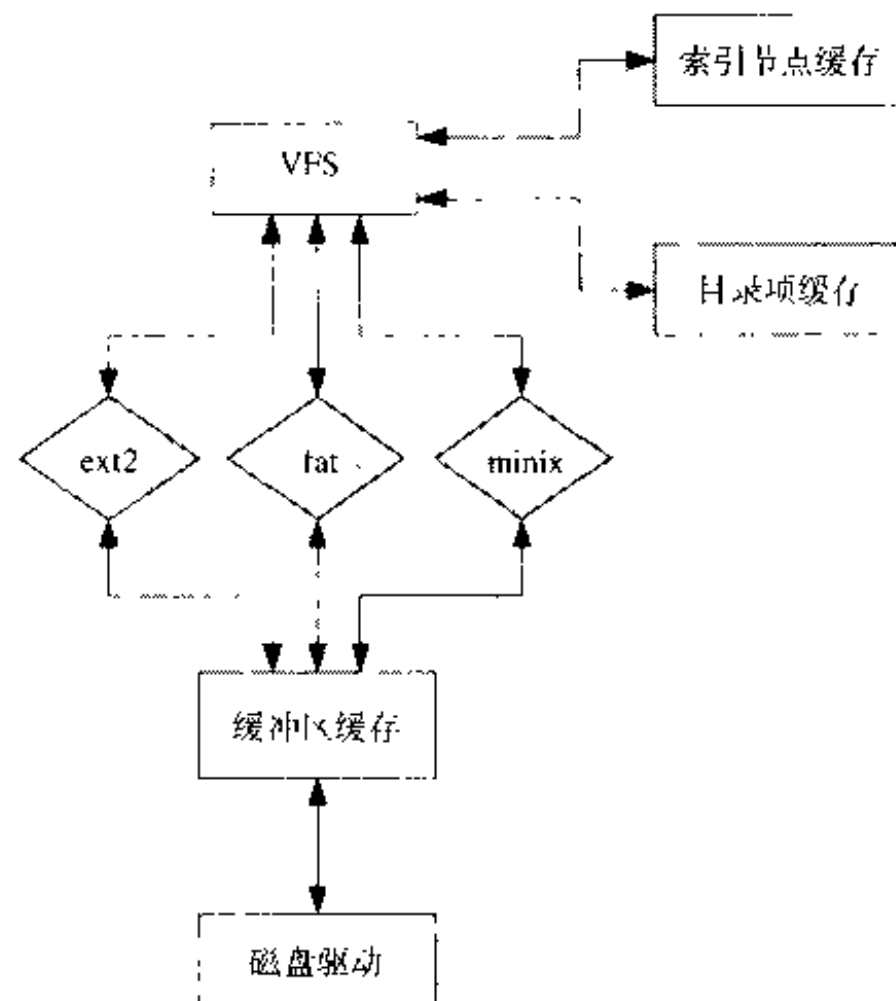


图5-13 Linux的VFS

VFS有两个接口：一个是与用户的接口；另一个是与特定文件系统的接口。VFS与用户的接口主要是V节点（Vnode，虚拟节点）。V节点是内核中的一个活动文件基类抽象。它定义了文件的访问接口，并且将所有对文件的操作定向到相应的特定文件系统函数上。VFS与特定文件系统的接口主要是通过vfsops来实现的。可以将vfsops作为一个通用的文件系统操作的抽象类，而每一种文件系统都应提供该类的一个具体实现。

下面通过Linux操作系统，来简要介绍VFS的工作机制，参见图5-13。Linux文件系统包括三部分：第一部分为VFS。这是Linux文件系统对外的接口。任何要使用文件系统的程序都必须经由这层来使用它。第二部分是高速缓存区。第三部分为真正最底层的具体文件系统，如ext2，vfat等。

Linux内核是通过VFS接口来使用文件系统的，因而Linux可以支持多个不同的文件系统，每

种文件系统表示一个VFS的通用接口。VFS作为实际文件系统（如EXT）和操作系统之间的接口，将它们隔离开。各文件系统通过为VFS提供一致的接口隐藏了文件系统的细节。由于VFS隐藏了Linux文件系统的所有细节，所以Linux核心的其他部分及系统中运行的程序将看到一个统一的文件系统。

进程在访问目录和文件的过程中，会调用系统函数对VFS节点进行遍历。由于每个文件和目录均由一个索引节点表示，因此，会重复访问部分索引节点。为了提高访问索引节点的速度，VFS将这些节点保存在索引节点高速缓存中。如果某个节点当前不在高速缓存中，则调用专用于某种文件系统的索引节点读取例程，读取该索引节点。频繁被读取的索引节点会保存在高速缓存中，而较少使用的VFS索引节点会从高速缓存中剔除。同时，VFS维护目录的高速缓存，以便能够快速找到频繁使用的目录索引节点。而目录高速缓存并不保存目录本身的索引节点，这些索引节点应当保存在索引节点高速缓存中；目录高速缓存实际保存的是完整目录名到对应索引节点编号的映射关系。

通过VFS，Linux可支持许多文件系统，例如ext2，minix，efs，bfs，devfs，proc，ramfs，romfs，isofs，udf，cramfs，vfat，hpfs，ntfs等。

3. Windows文件系统模型

在Windows 2000/XP中，I/O管理器负责处理所有设备的I/O操作。I/O管理器通过设备驱动程序、中间驱动程序、过滤驱动程序、文件系统驱动程序（File System Driver，FSD）等完成I/O操作，参见图5-14。

- 设备驱动程序：位于I/O管理器的最底层，直接对设备进行I/O操作。
- 中间驱动程序：与低层设备驱动程序一起提供增强功能。例如，当发现I/O失败，设备驱动程序可能简单地返回出错信息；而中间驱动程序却可能在收到出错信息后，向设备驱动程序发出再试请求。
- 文件系统驱动程序：扩展低层驱动程序的功能，以实现特定的文件系统，如NTFS。
- 过滤驱动程序：可以位于设备驱动程序与中间驱动程序之间，也可以位于中间驱动程序与文件系统驱动程序之间，还可以位于文件系统驱动程序与I/O管理器API之间。例如，一个网络重定向过滤驱动程序可以截取有关对远程文件的操作，并重定向到远程文件服务器上。

在以上组成构件中，与文件系统管理最为密切相关的当属FSD。FSD工作在内核模式中，但与其他标准内核驱动程序有所不同。FSD必须先要向I/O管理器注册，FSD还要与内存管理器与高速缓存管理器产生大量交互。因此，FSD使用了Ntoskrnl出口函数的超集。虽然普通内核设备驱动程序可以通过DDK（Device Driver Kit）来创建，但是对文件系统驱动程序则必须用IFS（Installable File System）来创建。

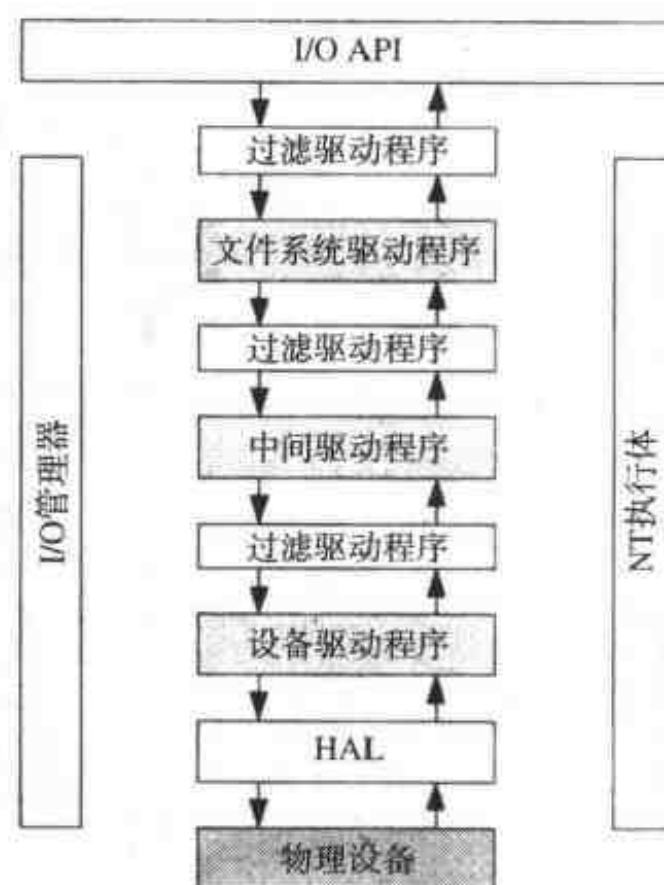


图5-14 Windows文件系统模型

4. 常用文件系统

随着操作系统的不断发展，越来越多的功能强大的文件系统不断涌现。这里，列出一些具有代表性的文件系统：

- sysv: System V/386, Coherent和Xenix文件系统。
- minix: 最老的Unix文件系统之一，相信也是最可靠的，但缺少特色(有些没有时间标记，文件名最长30个字符)，能力有局限(每个文件系统最多64 MB)。
- ext: ext2的老版，且不向上兼容。难于用新版安装程序安装，大部分人都改用ext2。
- ext2: Linux最为常用的文件系统，设计易于向后兼容，所以新版的文件系统代码无需改动就可以支持已有的文件系统。
- NFS: 网络文件系统，允许多台计算机之间共享文件系统，易于从所有这些计算机上存取文件。
- hpfs: IBM OS/2文件系统。
- FAT: FAT文件系统于1982年开始应用于MS-DOS中。经过了MS-DOS, Windows 3.x, Windows 9x, Windows NT, Windows 2000, Windows XP和OS/2等操作系统的不断改进，它已经发展成为包含FAT12, FAT16和FAT32的庞大家族。
- NTFS: NTFS是微软为了配合Windows NT 的推出而设计的文件系统，为系统提供了极大的安全性和可靠性，甚至有人认为它是当今世界上最出色的文件系统。后面的章节中，我们将对它进行深入的研究。

5.3.2 文件系统可恢复性

由于磁盘速度远远低于内存与处理器的速度，磁盘读写就成为计算机整体速度的瓶颈。对此，常常通过使用缓存来改善性能。这样，文件读写只要通过缓存读写，而不需要通过磁盘读写来完成。虽然文件读操作不会破坏磁盘一致性，但是由于文件写操作是间接通过缓存而不是直接通过磁盘来完成的，这样可能由于突然掉电等原因而导致文件系统的不一致。因此，不同的写操作方式直接影响着文件系统的可恢复性。

下面从文件系统的一致性和速度性能两个方面，来讨论一下文件系统的写入设计方式。这里主要分析最具代表性的三种写入方式：谨慎写（careful write）文件系统、延迟写（lazy-write）文件系统和事务日志（transaction log）文件系统。

1. 谨慎写文件系统

谨慎写文件系统因采用谨慎写而得名。谨慎写是对写入操作进行逐个排序的写入方式。当收到一个更新磁盘的请求时，会首先按一定顺序完成几项子操作，然后再进行磁盘的更新。例如，当为一个文件分配磁盘存储空间时，文件系统首先在它位图文件中设置位标记，然后再分配空间。在这个过程中如果磁盘发生中断，可能会出现文件系统的不一致，但是现有的数据不会被损坏。这样，通过运行专用的应用程序，在系统崩溃时所引起的卷错误是可以恢复的。

FAT文件系统使用类似的“通写”（write-through）技术，使得磁盘修改立即会写到磁盘上，而不需要把写操作排序以防止不一致性。

2. 延迟写文件系统

谨慎写虽然提供了对文件系统的可恢复性支持，但却是以牺牲速度为代价的。延迟写则通过利用“回写”(write back)高速缓存的方法获得了高速度。也就是说，系统只是把文件的修改写入高速缓存，然后在适当时候选用一种最佳方式把高速缓存的内容刷新到磁盘上。

众所周知，计算机的磁盘操作一般都是比较低速的，使用延迟写技术大大减少了磁盘操作的频率，从而极大地改善了系统的性能。但是，在系统崩溃时极有可能导致严重的磁盘不一致性，甚至于无法进行文件系统的恢复，因而具有一定的风险。

3. 可恢复文件系统

可恢复性文件系统试图既超越谨慎写文件系统的安全性，也达到延迟写文件系统的速度性能。可恢复性文件系统常采用事务日志来实现文件系统的写入。下面以NTFS为例，来简要讨论一下可恢复性文件系统。

NTFS通过基于事务处理模式的日志记录技术，成功保证了NTFS卷的一致性，实现了文件系统的可恢复性。事实上，NTFS的恢复过程是相当精确的，保证了卷能够被恢复到一致的状态。

当系统启动后文件记录服务就开始记录对所有文件的变动。这些记录包括文件的创建、打开、更新和其他操作。文件更新时系统会让高速缓存记录哪些文件变动了，同时事务日志将记录文件的更新操作。如果文件更新的磁盘操作失败了，事务日志就可以帮助恢复；如果文件更新的磁盘操作成功了，则事务日志也会被更新。

NTFS的可恢复性保证了卷结构不被破坏，从而保证了即使在系统失败时仍然可以访问所有的文件。虽然NTFS不能完全保证用户数据的安全——有些变动可能会从高速缓存中丢失，但是这些数据结构的改变都已被记录在日志文件里，文件系统的变动完全可以从日志文件中恢复。同时，NTFS对用户文件的记录有足够的可扩展性，用户可以通过FtDisk等工具来设置并保持冗余的数据存储。

当然，采用这些措施会付出代价，但是，代价可以通过高速缓存的延迟写技术来弥补，甚至可以增加高速缓存刷新之间的时间间隔。这样做不仅弥补了进行记录活动的系统耗费，有时甚至有所超越。

5.3.3 文件系统安全性

无论是对于个人用户还是企业级用户，都需要文件系统提供良好的安全性支持，对其中存储的文件进行有效的保护，以防止非法用户通过不正当途径取得机密数据。在国防、科研、银行、医院等敏感部门，对数据的安全性要求尤为突出。因此，在文件系统特别是大型文件系统的设计过程中，严格的安全保密措施是必不可少的。当今的大型文件系统，例如NTFS，主要采用两个措施来进行安全性保护：一是对文件和目录的权限设置，二是对文件和目录进行加密。

1. 文件与目录的权限

共享资源的安全性是所有操作系统都必须认真对待的问题。当今大部分高级文件系统都对文件和目录设置权限，对用户进行权限审核，以阻止非法用户的访问。在这样的文件系统中，文件拥有者在创建文件的时候就可以设定本人及其他用户对该文件的访问权限。这些权限信息可存放

在文件目录中，拥有者可以通过操作系统提供的命令随时进行修改。下面以NTFS为例来具体说明文件与目录的权限设置。

NTFS卷上的每个文件和目录在创建时创建人就被指定为拥有者。拥有者控制着文件或目录权限设置，并能赋予其他用户访问权限。

NTFS为了保证文件和目录的安全性和可靠性，制定了以下的权限设置规则：

- 只有用户在被赋予权限或是属于拥有这种权限的组，才能对文件或目录进行访问。
- 权限是累积的。如果组A用户对一个文件拥有“写入”权限，组B用户对该文件只有“读取”权限，而用户C同属两个组，则C将获得“写入”权限。
- “拒绝访问”权限优先级高于其他所有权限。如果组A的权限是“写入”，而组B是“拒绝访问”，那么同属两个组的用户C也不能读写文件。
- 文件权限始终优先于目录权限。
- 当用户在相应权限的目录中创建新的文件和子目录时，创建的文件和子目录继承该目录的权限。
- 创建文件或目录的拥有者，总是可以随时更改对文件或目录的权限设置来控制其他用户对该文件或目录的访问。

通过对文件和目录的权限设置，用户可以共享相应权限的文件数据，不仅为不同用户完成共同任务提供了基础，而且还节省了大量的磁盘空间。

2. 文件内容的加密

在信息交流高度发达的网络时代，很难防止非法用户对某些重要数据的窃取行为。因而对于高度机密的关键数据，除了设置权限以外，还需要对其进行加密，以更有效地保障其安全性。文件加密是对文件中的内容，按照一定的变换规则进行重新编码，从而得到新的无法正常可读的加密文件。

尽管加密技术在不断完善，但随着计算机计算速度的发展，绝对的无法解密的加密算法是不存在的。应该说，任何加密技术都是可以破解的。因而我们只是需要保障数据在一个特定时期内的安全性。即破解加密数据应该具有足够大的难度，以至短时间内破解是不可能的。

当今流行的加密算法有置换表算法、对称密钥算法以及非对称密钥算法等。

置换表算法是最简单的加密算法，但是也能很好的满足文件加密的需求。这种方法在计算机出现之前就已经被广泛的使用。文件中每一个数据段（一般一个字节）对应着“置换表”中的一个偏移量，偏移量所对应的值就输出成为加密后的文件。加密程序和解密程序都是通过这样一个置换表进行的。这种加密算法比较简单，加密解密速度都很快，但缺点也很明显：一旦这个“置换表”被对方获得，那这个加密方案就完全被识破了。

对称密钥算法则是加密解密都使用相同密钥的加密算法。不同的密钥可以产生不同的加密结果。这种算法一般速度比较快，适合大文件和数据的加密解密。但是缺点也和置换表算法一样：一旦丢失了密钥，那么文件系统的加密防护就形同虚设。

非对称密钥算法是加密解密使用不同的密钥的算法。著名的基于RSA（Rivest Shamir Adleman）的公共密钥算法（public/private）就是其中的一种。它通过一个公共密钥来加密文件，

而只有用户的私有密钥才能对该文件进行解密。在该算法的设计中，通过公共密钥来破解出私有密钥几乎是不可能的，因而具有极高的安全性。但是这种加密解密算法速度一般都比较慢，无法满足文件系统高速运作的要求。

在介绍NTFS的加密文件系统（Encrypted File System, EFS）时，我们将更深入地探讨这个问题。

5.4 Windows FSD体系结构

Windows 2000/XP的FSD（File System Driver，文件系统驱动程序）可分为本地FSD和远程FSD。前者允许用户访问本地计算机上的数据；而后者则允许用户通过网络访问远程计算机上的数据。

5.4.1 本地FSD

本地FSD包括Ntfs.sys, Fastfat.sys, Udfs.sys, Cdfs.sys, Raw FSD等，参见图5-15。本地FSD负责向I/O管理器注册自己。当开始访问某个卷时，I/O管理器将调用FSD来进行卷识别。

对于Windows 2000/XP所支持的文件系统，每个卷上的第一个扇区都是作为启动扇区而预留的。启动扇区虽然只有512字节，但是却包含有足够多的信息以供确定卷上文件系统的类型和定位文件系统元数据的位置。另外，卷识别常常需要对文件系统进行一次检查。

当完成卷识别后，本地FSD还创建一个设备对象以表示所装载的文件系统。I/O管理器也通过卷参数块（Volume Parameter Block, VPB）为由存储管理器所创建的卷设备对象和由FSD所创建的设备对象之间建立连接。该VPB连接将I/O管理器的有关卷的I/O请求转交给FSD设备对象。

本地FSD常常用高速缓存管理器来缓存文件系统的数据（如元数据），以提高性能。本地FSD与内存管理器一起实现内存映射文件。本地FSD也支持文件系统卸载（unmount）操作，以便提供对卷的直接访问。当应用程序需要通过文件系统访问数据时，I/O管理器将重新装载（mount）文件系统。

5.4.2 远程FSD

远程FSD由两部分组成：客户端FSD与服务器端FSD。客户端FSD允许应用程序访问远程的文件和目录。客户端FSD首先接收来自应用程序的I/O请求，接着转换为网络文件系统协议命令，然后再通过网络发送给服务器端FSD。服务器端FSD监听网络命令，接收网络文件系统协议命令

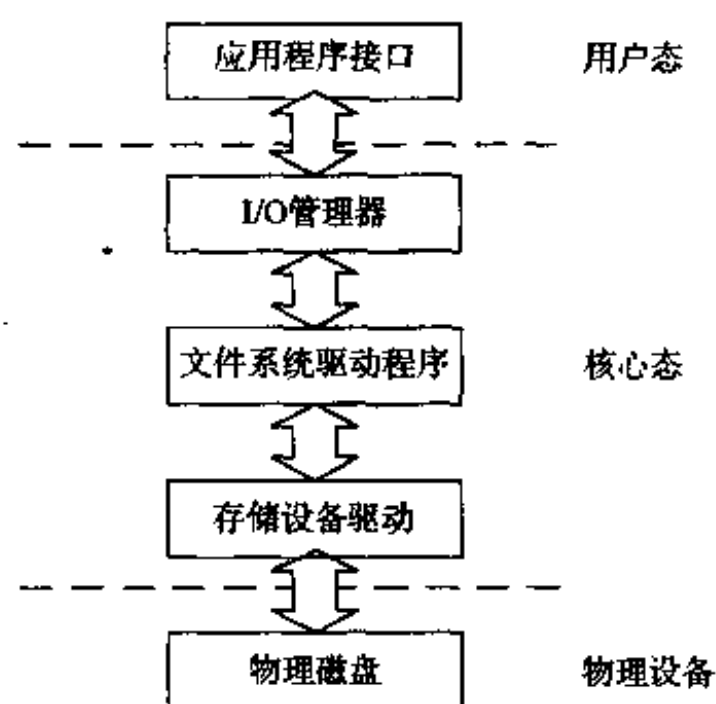


图5-15 本地FSD

并转交给本地FSD去执行。关于客户端FSD与服务器端FSD之间的交互，参见图5-16。

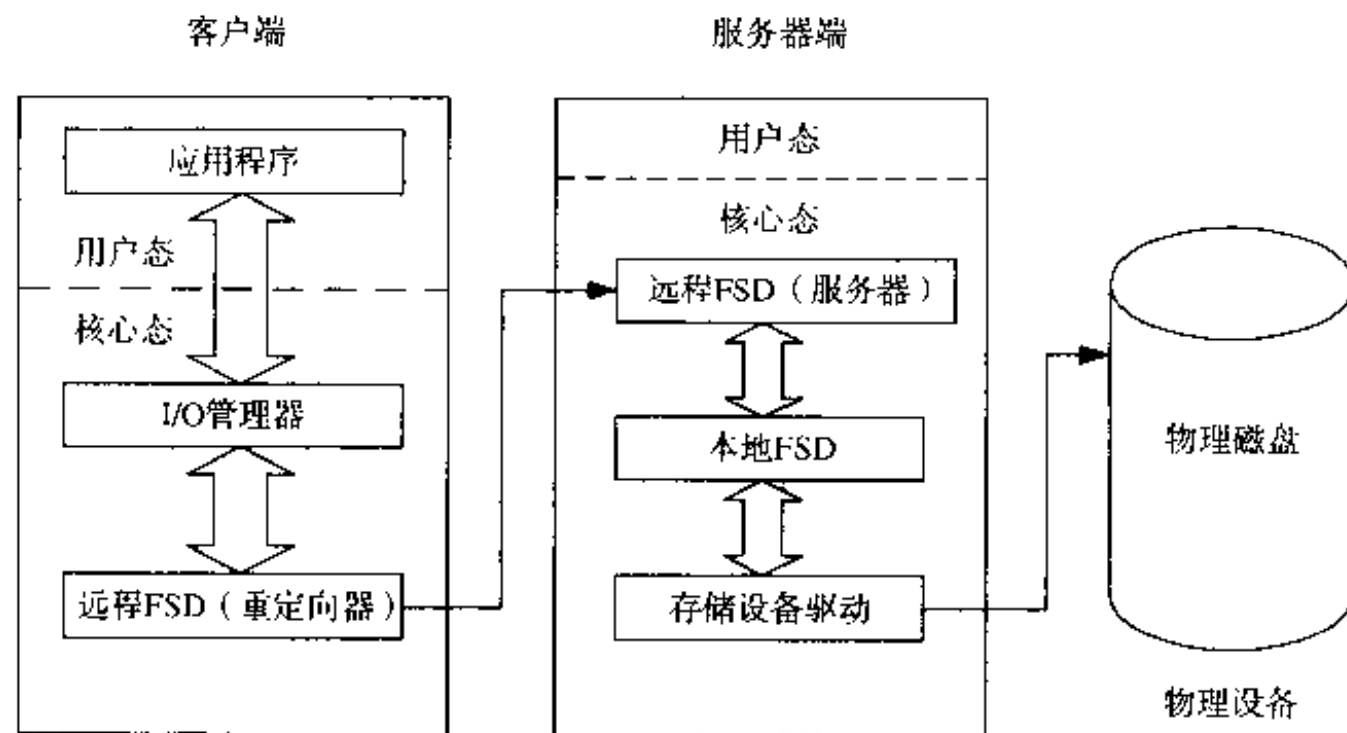


图5-16 远程FSD

对于Windows 2000/XP而言，客户端FSD为LANMan 重定向器（LANMan Redirector），而服务器端FSD为LANMan 服务器（LANMan Server）。重定向器是通过端口/小端口（port /miniport）驱动程序组合来实现的，其中端口驱动程序（\Winnt\System32\Drivers\Rdbss.sys）实现为驱动程序函数库，而小端口驱动程序（\Winnt\System32\Drivers\Mrxsmbs.sys）是通过使用端口驱动程序的服务来实现的。

Windows 2000/XP通过通用互联网文件系统（Common Internet File System、CIFS）协议来实现重定向器与服务器之间的通信。CIFS是Microsoft服务器消息块（Server Message Block、SMB）协议的改进版本。

5.4.3 FSD与文件系统操作

Windows文件系统的有关操作都是通过FSD来完成的，参见图5-17。具体地说，有如下几种方式会用到FSD：显式文件I/O、高速缓存延迟写、高速缓存提前读、内存脏页写与内存缺页处理。下面，作一简要介绍：

- 显式文件I/O：应用程序通过Win32 I/O接口函数如CreateFile，ReadFile及WriteFile等来访问文件。函数CreateFile是通过Win32客户端DLL Kernel32.dll来实现的。函数CreateFile通过NtCreateFile来完成。NtCreateFile通过ObOpenObjectByName解析名称字符串，通过IoParseDevice创建I/O请求包（I/O request packet，IRP），通过IoCallDriver将IRP交给合适的FSD以创建文件。函数ReadFile通过NtReadFile来完成。NtReadFile将已打开文件的句柄转换成文件对象指针，检查访问权限，创建IRP读请求，通过IoCallDriver将IRP交给合适的FSD。如果文件可以放在高速缓存，那么需要检查PrivateCacheMap：如有效则表示该文件已有私有高速缓存映射结构；如无效则表示尚没有私有高速缓存映射结构，需要调用

CcInitializeCacheMap来初始化。NtReadFile通过CcCopyRead从高速缓存中读取数据。如果数据还不在于高速缓存中，CcCopyRead会引起缺页中断，并间接调用MmAccessFault。函数WriteFile与ReadFile相类似；只不过WriteFile调用NtWriteFile，且FSD调用CcCopyWrite而不是CcCopyRead。

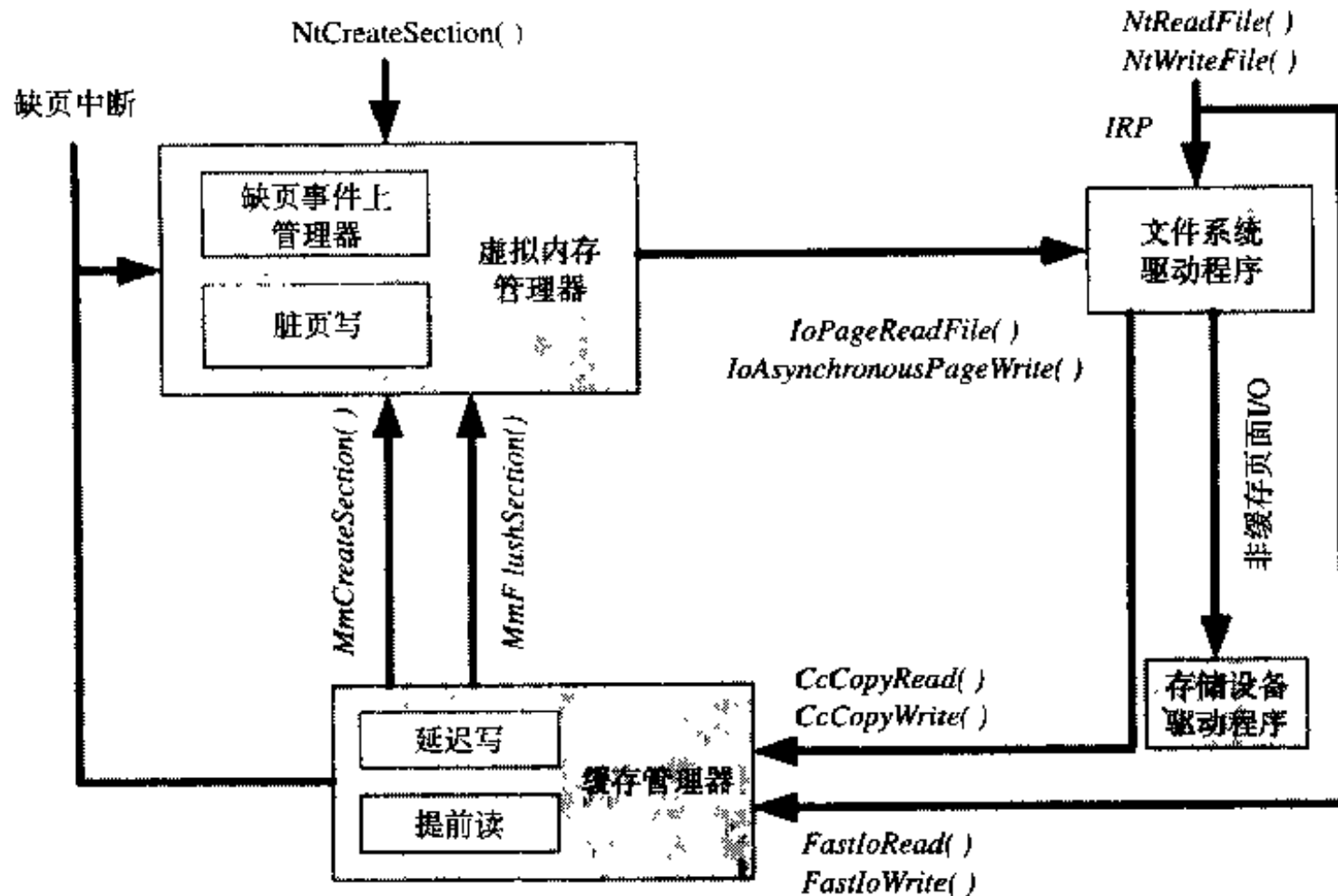


图5-17 FSD的作用

- **高速缓存延迟写：**高速缓存管理器的延迟写线程定期地对高速缓存中已被修改的页面进行写操作。这是通过调用内存管理器的MmFlushSection函数来完成的。具体地说，MmFlushSection通过IoAsynchronousPageWrite将数据送交FSD。
- **高速缓存提前读：**高速缓存管理器的提前读线程负责提前读数据。提前读线程通过分析已作的读操作，来决定提前读多少。提前读线程是通过缺页中断来完成的。
- **内存脏页写：**内存脏页写线程定期地清洗缓冲区。该线程通过IoAsynchronousPageWrite来创建IRP写请求，这些IRP被标识为不能通过高速缓存，因此它们被FSD直接送交到磁盘存储驱动程序。
- **内存缺页处理：**以上在进行显式I/O操作与高速缓存提前读时，都会用到内存缺页处理。另外，只要应用程序访问内存映射文件且所需页面不在内存时，也会产生内存缺页处理。内存缺页处理MmAccessFault通过IoPageRead向文件所在文件系统发送IRP请求包来完成。

5.5 Windows文件系统概述

Windows 2000/XP直接支持以下几种文件系统:

- CDFS与UDF
- FAT12, FAT16与FAT32
- NTFS

这里首先简要介绍CDFS与UDF,然后再较详细地描述FAT12, FAT16与FAT32。在本节之后将对NTFS进行深入分析。

5.5.1 CDFS与UDF

只读光盘文件系统 (CDROM File system, CDFS) 是1988年为只读光盘所制定的文件系统标准。CDFS比较简单,但是有一定的限制:

- 文件和目录名的长度必须少于32个字符。
- 目录树的深度不能超过8层。

CDFS现在已过时,已被UDF标准所代替。但是为了向后兼容,Windows 2000/XP仍提供对CDFS的支持,这是通过\Winnt\System32\Drivers\Cdfs.sys实现的。

通用磁盘格式 (Universal Disk Format, UDF) 是于1995年由光学存储技术协会 (Optical Storage Technology Association, OSTA) 为光磁盘存储媒介如DVD-ROM等所制定的,用来代替CDFS,比CDFS更加灵活。UDF具有如下特点:

- 文件名区分大小写
- 文件名可以有255字符长
- 最长路径为1023个字符

Windows 2000/XP通过\Winnt\System32\Drivers\Udfs.sys来实现对UDF的支持。

5.5.2 FAT12、FAT16与FAT32

文件分配表 (File Allocation Table, FAT) 文件系统属遗留文件系统。但是为了向后兼容,也为了方便用户升级,Windows 2000/XP仍然提供对FAT的支持。这是通过\Winnt\System32\Drivers\Fastfat.sys来提供FAT文件系统驱动程序的。下面分别介绍FAT的三种形式: FAT12, FAT16与FAT32。

1. FAT12与FAT16

每一种FAT文件系统都用一个数字来标识磁盘上簇号的位数。例如, FAT12的簇标识为12位 (二进制数), 这限制了它的单个分区最多只能存储 2^{12} (=4096) 个簇, 而FAT 12在Windows 2000/XP中的簇大小在512 B与8 KB之间, 这意味着FAT12卷的大小至多只有32 MB。Windows 2000/XP使用FAT12来作为5.25英寸 (1.2 MB) 和3.5英寸 (1.44 MB) 软盘的标准格式。

FAT16文件系统自1982年开始应用于MS-DOS中。它的16位簇标识符使其能够定位65536个簇。FAT16在Windows 2000/XP中的簇大小从512 B到64 KB, 这也使它的卷大小理论上可以达到4 GB, 但Windows操作系统把它限制在2 GB以内。在Windows 2000/XP中它的卷大小和簇的变化关系列在表5-1中:

表5-1 FAT16的簇大小

卷大小	簇大小
0 ~ 32 MB	512 B
33 ~ 64 MB	1 KB
65 ~ 128 MB	2 KB
129 ~ 256 MB	4 KB
257 ~ 511 MB	8 KB
512 ~ 1023 MB	16 KB
1024 ~ 2047 MB	32 KB
2048 ~ 4095 MB	64 KB

要注意的是，如果我们格式化了一个0 ~ 32 MB的分区，Windows 2000/XP会使用FAT12而不是FAT16。在磁盘的实现上，一个FAT卷分为数个区域，参见图5-18。

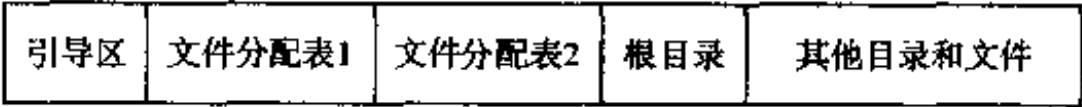


图5-18 FAT卷的结构

文件分配表包含一个卷上所有簇的条目，对于保持卷的完整性具有重要意义，因此为它保留了一个备份。当系统驱动或者是一致性检查程序不能访问分配表时，就可以读取备份信息。

文件分配表的条目还定义了文件分配链（File Allocation Chain）来连接文件和目录。链上的链接是文件存储的下一个簇的标号。文件分配链的结尾被指定为0xFFFF（FAT16）或是0xFFF（FAT12）。未使用的簇则被标为0。图5-19清楚地说明了这一点：

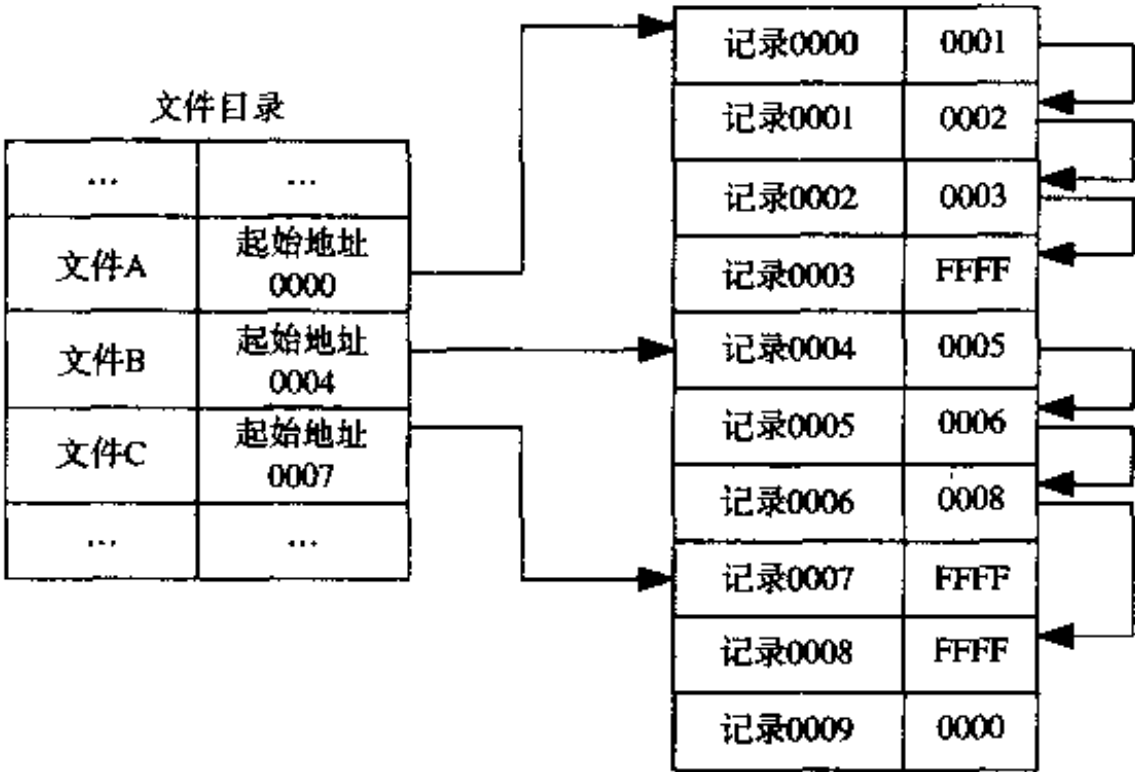


图5-19 FAT文件分配链示意图

FAT12和FAT16的根目录预留了足够的空间来存储256个目录项，这意味着根目录最多只能存

放256个文件或目录（而FAT32则无此限制）。一个FAT目录项包含32个字节来存放文件的名称、大小、开始簇以及时间戳（包括上次的打开，创建等的时间）。

FAT16文件系统的主要优点是它可以被多种操作系统访问，如MS-DOS，Windows 3.x，Windows 9x，Windows NT和OS/2等。但是，FAT16文件系统不支持长文件名，人们给文件命名时也要受8个字符名和3个字符的扩展名的8.3命名规则的限制。同时，FAT16文件系统无法支持系统高级容错特性，不具有内部安全特性等。

在Windows 9x中，通过对FAT文件系统的扩展，长文件名问题得到了解决，这也就是人们所说的扩展FAT文件系统，文件名可长达255个字符，所以人们很容易通过文件名来表现文件内容。但是为了同MS-DOS和Win16程序兼容，它仍保留有扩展名。同时它也支持文件日期和时间属性，为每个文件保留了文件创建日期/时间、文件最近被修改的日期/时间和文件最近被打开的日期/时间这三个日期/时间戳。

2. FAT32

FAT32是微软最新定义的基于FAT模式的文件系统，是对早期FAT16文件系统的增强。由于文件系统的核心——文件分配表簇标识由16位扩充为32位，所以称为FAT32文件系统，主要应用于Windows 9x以及Windows Me系统。在FAT16文件系统中单个磁盘空间的大小不能超过2 GB，分区时单簇所占的磁盘空间比较大，如果磁盘大于10 GB，使用FAT32能使磁盘得到充分利用。由于FAT32高达32位的簇标识符（它的高4位被暂时保留，所以真正有效的是28位），簇大小也能达到32 KB，这使FAT32理论上拥有8 TB的惊人寻址能力，而Windows 2000/XP则限制它的卷大小为32 GB。FAT32强大的寻址能力使它比FAT16能够更有效地管理磁盘。

FAT32的卷与簇的大小关系如表5-2所示：

表5-2 FAT32的簇大小

卷 大 小	簇 大 小
32 MB ~ 8 GB	4 KB
8 GB ~ 16 GB	8 KB
16 GB ~ 32 GB	16 KB
32 GB	32 KB

在磁盘分区大小超过512兆字节时使用这种格式，会有效地存储数据，减少磁盘空间的浪费，还可加快程序的运行，使用的计算机系统资源更少，因此是使用大容量磁盘存储文件的有效文件系统。

FAT32的根目录区（ROOT区）不再是固定区域、固定大小，而是可以看作数据区的一部分。因为根目录已改为根目录文件，采用与于目录文件相同的管理方式，因此根目录下的文件数目不再受最多256的限制。目录区中的根目录项变化较多，一个目录项仍占32字节，可以是文件目录项、子目录项、卷标项（仅根目录有）、已删除目录项、长文件名目录项等。

由于FAT32驱动器中的引导记录被扩展为包含重要数据结构的备份，这使得FAT32分区比现存的FAT16不易受到单点的错误影响，从而大大提高了可靠性。FAT32驱动器上的根目录现在也成了

一个普通的簇链，因而可以放到驱动器的任何地方。由于这个原因，以前根目录项数的限制就不复存在了。它也支持长文件名格式，但仍保留有扩展名。但是FAT32文件大小也要求在4 GB以下，同样无法支持高级容错特性，不具有内部安全特性，因而仍然无法达到高性能文件系统的要求。

5.6 NTFS设计目标与高级特性

5.6.1 NTFS设计目标

在Windows NT 3.1推出前，Microsoft就发展了两种文件系统：基于MS-DOS和MS Windows的FAT（File Allocation Table）文件系统以及用于OS/2操作系统的高性能文件系统（High Performance File System, HPFS）。伴随Windows NT的推出，微软需要一种新的文件系统来支持NT的安全性和可靠性，而FAT和HPFS显然在此方面存在先天缺陷。经过慎重考虑，NT设计小组决定创建一种具有较好容错性和安全性的全新的文件系统——NTFS（New Technology File System）。从Windows NT 3.1一直到Windows 2000/XP，NTFS不断地得到发展，相对与FAT和HPFS来说，NTFS在取得了令人羡慕的性能特征和向后兼容性的同时，成为第一个为高端服务器以及Intel工作站家族提供健全的文件服务的文件系统。

从一开始，NTFS就定位为企业级的文件系统。为了减少因突然电源掉电或系统发生崩溃所造成的数据丢失，文件系统应始终确保文件系统元数据的完整性；为了保护敏感数据免受非法访问，文件系统应有一个综合的安全模型；为了保护用户数据，文件系统应提供廉价的基于软件的数据冗余方案以替代较为昂贵的基于硬件的数据冗余方案。

1. 可恢复性

相对个人用户来说，最关心的是磁盘I/O性能即磁盘I/O速度，但对于Windows 2000/XP的服务器市场来说，数据存储的可靠性则更为重要。如果磁盘I/O由于各种原因失效，那么不仅操作的速度毫无意义，甚至可能造成无法挽回的严重损失。

为了满足可靠数据存储和访问的需求，NTFS提供了基于原子事务（Atomic transaction）概念的文件系统可恢复性。原子事务是数据库中处理数据更新的一项技术，它能保证数据库的正确性和完整性不受系统失败的影响。原子事务的基本原则就是称为事务的数据库操作要么都做要么都不做。如果系统失败中断了事务，则已经做的部分要撤消或回退（Rollback）。回退可以将数据库返回到先前的稳定状态，就像是失败的事务从来也没有发生过一样。

NTFS采用这种事务处理模式来实现文件系统可恢复性。如果一个程序启动了一个改变NTFS结构的I/O操作，如改变目录结构、增长文件、增加新文件等等，那么NTFS都将该操作视为一个原子事务。

另外，NTFS对关键文件系统信息还采用了冗余存储。这样，即使磁盘上的某个扇区损坏，NTFS仍可以访问卷上的关键数据。这种磁盘上文件系统数据的冗余存储与FAT和HPFS是不一样的，后者在磁盘上只有单个关键数据，因此如果不能读取这些关键数据，则整个卷的数据就会丢失。

2. 安全性

当涉及到个人隐私或者是某些敏感部门——例如银行、证券以及国防机构的数据时，数据安全性显得尤为重要，用户必须具备相应的权限才能访问数据。Windows 2000/XP 如果想要进军上述市场，必须对此具有很好的支持。

与NTFS文件系统相结合，Windows 2000/XP系统能够指定谁能访问某一文件或目录和对它进行什么操作。在创建一个文件时，可以通知Windows 2000/XP，哪些用户可以读该文件，哪些用户可以修改该文件；另外，还可以指定谁可以列出一个目录的内容和谁可以在该目录下增加文件。即使用户知道文件的路径，仍可以禁止访问目录中的文件。

NTFS的文件系统不易受到病毒和系统崩溃的侵袭，这种抗干扰直接源于Windows 2000/XP操作系统的高度安全性能。即使在FAT 和NTFS两种文件系统在一个磁盘中并存时，由于NTFS文件系统只能被 Windows 2000/XP识别，一般的病毒还是很难在NTFS文件系统中找到生存空间。

NTFS的安全性直接来源于Windows XP的对象模型。NTFS的基本思想是：把文件和目录看成是对象和对象的集合。目录的内容不必受到下层的文件系统存储机制的束缚，可以把它们作为独立的实体来访问与复制。文件和目录对象都带有安全描述符，这些描述符作为该文件的一部分存储在磁盘上。进程在打开任何对象（如文件对象）的句柄前，Windows 2000/XP安全系统就验证该进程是否具有足够的权限。安全描述符和用户登录到系统并提供识别的密码结合起来共同确定了该进程的权限，从而保证了除非有管理员或文件拥有者的授权，否则无法访问文件。

NTFS还支持加密文件系统（Encrypted File System, EFS），可以阻止非授权用户访问加密文件。

3. 数据冗余和容错

有些用户不但需要文件系统数据的可恢复性，而且还需要保证他们的数据不会因电源掉电或灾难性磁盘故障而受损。虽然NTFS的可恢复性可以确保能够访问某个卷上的文件系统，但是却不能保证完全恢复用户文件。为了保护文件数据，可以采用数据冗余存储。

Windows 2000/XP采用分层驱动器模型实现了数据冗余存储，提供了数据的容错性支持。NTFS与卷管理器通信，而后者又与磁盘驱动程序通信以便将数据写入磁盘。卷管理器能够映射或复制一个磁盘的数据到另一个磁盘，因此一个冗余拷贝总是可以获得的。这种支持通常称为RAID-1。卷管理器也允许将数据按条写入到三个或更多的磁盘上，有关数据校验信息也保存在某个磁盘上。如果一个磁盘上的数据受损或无法访问，则可以通过异或操作来恢复。这种支持称为RAID-5。

5.6.2 NTFS的高级特性

为了适应众多应用领域，NTFS不但满足了其基本设计目标如可恢复性、安全性和数据冗余与数据容错，而且还具有其他一系列高级特性，如多数据流、基于Unicode的名称、通用索引机制、动态坏簇重新映射、硬链接、文件压缩、日志记录、磁盘限额、链接跟踪、加密、POSIX支持、碎片整理等。下面将简要介绍这些高级特性。

1. 多数据流

在NTFS中，与文件相关的每个信息单元，包括文件名、文件的拥有者、文件的时间标记、

文件的内容等，都是当做NTFS对象属性（NTFS Object Attribute）来实现的。这种统一实现便于向文件增加更多属性。因为文件的数据仅仅是一种属性且可以增加更多属性，所以NTFS文件可以包含多个数据流。

每个流都有其各自的分配大小（已预留的磁盘空间），实际大小（实际使用了多少字节空间），以及有效的数据长度（初始化了多少数据流），等等。另外，每个流都有一个单独的文件锁，用来锁定一定范围的字节并允许并发访问。为了降低处理开销，每个文件共享可共享文件锁而不是让每个文件的所有流都使用不同的文件锁。

NTFS文件有一个缺省数据流，该流没有名称。应用程序可以创建其他的具有名称的数据流，且可通过指定名称来访问这些数据流。为了避免改变用字符串作为文件名参数的Win32 I/O API，可以通过在文件名后先加上“:”再加上数据流名称来完成。这是因为冒号是保留字符，可用来作为文件名和数据流名之间的分隔符，例如：

Mytext.txt:Stream2

NTFS所特有的多数据流文件为许多应用程序（如高端服务器应用程序）提供了一种创造性的解决手段，例如支持Apple Macintosh文件系统等。但是，也应当注意，当混用NTFS和非NTFS文件系统时，也会造成一些兼容性的问题。例如，当你拷贝一个多流文件到非NTFS卷时，只拷贝了主要的流。这意味着你丢失了额外的数据，即使你再次拷贝回NTFS卷，它们也不能被恢复。

Windows 2000/XP 自带的Apple Macintosh文件服务器支持就使用多数据流。Macintosh的每个文件都使用两个数据流：一个用于存储数据，另一个用于存储资源信息，如文件类型和文件图标等。因为NTFS允许多数据流，所以一个Macintosh用户可以将整个Macintosh目录拷贝到Windows 2000/XP，而另一个Macintosh用户可以在不丢失信息的前提下，将这一拷贝再复制到本机上。

Windows资源管理器也使用多数据流。当右击一个NTFS文件并选择属性时，所产生的对话框中的总结部分可以让你为该文件关联一些信息，如标题、主题、作者及关键词等。Windows资源管理器将这些信息作为另一个名为总结信息的流，加到文件中去。其他程序也使用多数据流这一特点。例如，一个备份程序可以用一个额外的数据流来存储特定时间信息。

2. 基于Unicode的名称

如整个Windows 2000/XP一样，NTFS完全支持Unicode，完全使用Unicode字符来存储文件、目录和卷的名称。Unicode是一种16位的字符编码方案，世界上每种主要语言中的每个字符都能够被唯一地表示，这有助于国际化。Unicode与传统的国际字符相比是一种改进，传统的做法是有的字符是8位，而有的是16位，且还需要加上编码表才可确定字符。因为Unicode对每个字符都有唯一地表示，所以不需要使用编码表。NTFS路径名中的每个文件名或目录名的长度可达255个字节，其中可以包含Unicode字符、多个空格及多个圆点。

此外，由于NTFS文件系统支持长文件名，人们给文件命名时也不必受8.3命名规则限制，从而可以给文件起一个反映其意义的文件名。NTFS支持向后兼容，甚至可以从新的长文件名中产生老式的短文件名。当文件写入可移动存储介质(如软盘)时，它自动采用FAT文件名和FAT文件系统。

3. 通用索引机制

NTFS文件系统能够在磁盘卷中索引文件属性，从而大大提高了文件管理的效率。这使得文件系统能够有效地定位匹配某种条件的文件。例如，查找同一个拥有者的文件将会十分方便。FAT文件系统仅仅索引文件名而不把它们分类，这使得在大目录下查找文件十分费时。

许多NTFS的特点都充分使用了通用索引，如综合安全描述符。将卷上的文件和目录的安全描述符存储于同一个内部流，再删除重复部分，就可按照NTFS所定义的内部安全标识来建立索引。

4. 动态坏簇重印映射

对于普通文件系统来说，如果用户试图从坏扇区中读取数据，那么读写操作失败，该扇区所在分配簇中的数据也将无法访问。而对于NTFS文件系统来说，容错驱动程序会将受损扇区的数据写入其他好的扇区，并标记具有坏扇区的簇的地址，防止以后再使用它。这对执行磁盘读写的任何应用程序是透明的。该特性有时叫做“热修复”(hot fix)。如果容错驱动程序在扇区损坏时没有被加载，NTFS仍然可以替换坏簇并不再使用它，但是不能恢复坏扇区中的数据。

5. POSIX支持

Windows 2000/XP完全支持可移植操作系统接口(Portable Operating System Interface) POSIX 1003.1。在文件系统方面，NTFS实现了POSIX 1003.1的所有要求。例如：

- 大小写敏感的文件名：在POSIX环境下，文件名是大小写敏感的。因而README.TXT, Readme.txt, readme.txt是不同的文件。
- 通过许可：当判定一个用户是否可以访问一个文件或目录时，需要考虑路径上的所有目录的安全许可。
- 文件改变时间：提供文件最后被访问的时间标记。
- 硬链接：在不同的目录下不同文件名的两个文件指向相同的数据时，两个文件发生硬链接。

6. 文件压缩

NTFS支持文件数据的压缩。因为NTFS可以透明地执行压缩和解压缩，应用程序不必修改就可利用这一特点。目录也可以压缩，这指该目录中以后所建文件均会被压缩。文件压缩能将文本性质的应用程序代码和数据文件压缩大约50%，将可执行文件压缩大约40%。

应用程序通过向DeviceIOControl传递FSCTL_SET_COMPRESSION文件控制代码来压缩或解压缩；通过传递FSCTL_GET_COMPRESSION文件控制代码来获取文件和目录的压缩状态。另外，应用程序也可以通过GetFileAttributes来确定文件和目录的压缩属性，这是因为压缩文件或目录的属性FILE_ATTRIBUTE_COMPRESSED会被置位。

NTFS压缩功能的使用将会引起NTFS卷的性能下降，原因是每次访问被压缩的文件时，都需要对它进行解压缩。如果要拷贝一个压缩文件，其过程是：解压缩、拷贝、重新对拷贝的文件进行压缩，大大增加了CPU的处理时间。

压缩的另外一种形式称为稀疏文件。如果一个文件被标记为稀疏，那么NTFS不会为被应用程序设置为0的部分分配空间。当应用程序从稀疏文件的空白区域读数据时，NTFS将返回用0填充的缓冲区。

与压缩文件一样，NTFS透明地管理稀疏文件。应用程序通过向DeviceIOControl传递FSCTL_SET_SPARSE文件系统控制代码来指定文件的稀疏状态。应用程序可以通过使用

FSCTL_SET_ZERO_DATA控制代码来将文件某部分设置为空，通过使用FSCTL_QUERY_ALLOCATED_RANGES控制代码来获取文件的哪些区域为空。稀疏文件的应用之一就是NTFS的日志文件。

7. 日志记录

许多类型的应用需要监视卷上文件或目录的改变。例如，一个自动备份的程序可能在刚开始时进行一个完全的备份，以后就在此基础上根据文件的变化进行小规模备份。实现这个功能的一个最简便的方法就是对卷进行扫描以记录文件和目录的状态，在以后的扫描时记录变化情况。然而，该进程会严重降低系统的性能，对具有庞大文件数目的系统影响更大。

另一方法是让应用程序通过Win32函数FindFirstChangeNotification或ReadDirectoryChangesW来注册目录通知。应用程序将需要监视的目录作为输入参数，只要目录改变时，函数就返回。这种方法比卷扫描的效率要高，不过它要求应用程序要一直运行。使用这些函数也需要应用程序扫描目录，这是因为FindFirstChangeNotification并不说明什么改变，而只说目录中有的东西改变了。应用程序可以向函数ReadDirectoryChangesW传递一个缓冲区域以便FSD可以记录变化。如果缓冲区域填满溢出，那么应用程序仍必须扫描目录。

为了克服以上两种方法的缺点，NTFS提供了第三种方法：应用程序可以向函数DeviceIOControl传递文件系统控制代码FSCTL_CREATE_USN_JOURNAL，来配置NTFS日志记录，这样NTFS将文件和目录改变记录到一个内部日志文件中。日志文件足够大，几乎可以保证应用程序能有机会来处理记录。应用程序可以使用FSCTL_QUERY_USN_JOURNAL文件控制代码来读日志文件，也可以指定只有新记录时DeviceIOControl才完成。

8. 磁盘限额

不同于个人计算机，服务器应用需要系统管理员能够管理分配给各个用户不同的磁盘空间。合法用户只能访问属于自己的文件，这对于提高服务器的安全性和可靠性是必要的。微软以前的文件系统对此无能为力，NTFS的设计满足了这个要求。NTFS可以支持用户磁盘限额。当用户超过其警告线时，NTFS可以记录这一事件。类似地，当用户试图使用超过其限额的空间时，NTFS也可以记录这一事件并让这一企图失败。

NTFS根据文件和目录的SID来跟踪记录用户的磁盘使用情况。在与限额比较时，NTFS使用文件和目录的逻辑大小来控制空间的使用。因此用户无法通过压缩文件和稀疏文件来使用超过其限额的空间。

缺省时，不使用磁盘限额跟踪。如果要使用，可以通过卷属性来完成。

9. 硬链接与软链接

硬链接允许从多个路径来指向同一个文件和目录。如果已有一个文件为C:\Home\Doc\NtBook\Chap5.doc，且为其创建一个硬链接为C:\chap5.doc，那么这两个路径指向同一文件，因此可以通过任一路径来对该文件进行操作。进程可以用Win32 API函数CreateHardLink或POSIX命令ln来创建硬链接。软链接允许用来重定向一个目录。软链接是基于重解析点的。

10. 链接跟踪

外壳快捷允许用户通过链接将文件加入到外壳名字空间。Windows 2000/XP的开始菜单使用

了很多外壳快捷方式。类似地，对象链接和嵌入（Object Linking and Embedding, OLE）技术允许将一个程序的文档透明地嵌入到其他程序中。Microsoft Office 2000套件，如PowerPoint, Excel, Word, 都使用了OLE技术。

虽然外壳和OLE提供了一种简单的方法将文件链接起来，但是这些链接却不好管理。如果Windows NT 4, Windows 95或Windows 98用户移动外壳和OLE的源文件，那么链接将会破坏，而系统只能靠启发来试图定位链接源。Windows 2000/XP提供称为分布链接跟踪的服务程序。该程序可以用来对移动源进行跟踪，以保持外壳和OLE链接的完整性。

NTFS链接跟踪是基于一个称为对象标识（Object ID）的可选文件属性。应用程序可以用FSCTL_CREATE_OR_GET_OBJECT_ID和FSCTL_SET_OBJECT_ID文件系统控制代码来为文件赋予一个对象标识；可以用FSCTL_CREATE_OR_GET_OBJECT_ID和FSCTL_GET_OBJECT_ID文件系统控制代码来查询对象标识；可以用FSCTL_DELETE_OBJECT_ID文件系统控制代码来删除对象标识。

11. 加密

企业级用户常常需要在计算机上存储敏感数据。虽然企业服务器上的数据常常有合适网络安全设置和物理访问保护，但是如果磁盘丢失，那么其上所存储的数据将会暴露。NTFS文件权限许可并不能保护数据。这是因为另一个装有Windows 2000/XP的机器的系统管理员将可以访问该文件；另外其他操作系统如Linux也可以不管权限而能读NTFS卷上的数据。

NTFS包含有一个称为EFS（Encrypting File System）的工具，可以用来加密数据。EFS操作如同文件压缩一样，对应用程序而言是透明的。也就是说，当授权用户的程序需要读数据时，数据自动解密；当需要改变数据时，数据自动加密。

EFS依赖于Windows 2000/XP所提供的处于用户态的加密服务。因此，EFS不但包括处于内核态的与NTFS紧密相关的设备驱动程序，而且还包括处于用户态的与LSASS（Local Security Authentication Subsystem，本地安全验证子系统）通信的DLL和加密DLL。

加密文件只能通过一个账号的EFS私有/公共密钥对的私有密钥来访问，而私有密钥用账号的口令加锁。因此，没有受权账号的口令，不能用其他方法访问EFS加密文件。

应用程序可以用Win32 API EncryptFile和DecryptFile来加密或解密文件；可以用FileEncryptionStatus来获得有关文件和目录的EFS有关属性，如文件和目录是否已加密。

12. 碎片整理

如果文件数据占据多个非连续的扇区簇，那么该文件就成碎片。与其他文件系统一样，NTFS并不试图确保文件为连续的。但是，主控文件表（Master File Table, MFT）是个例外，系统通常为MFT保留一些区域。因此，MFT通常是连续的。

Windows 2000/XP本身带有一个碎片整理工具。该碎片整理工具有一定限制，为了便于第三方开发碎片整理工具，Windows 2000/XP提供了碎片整理函数，用于移动文件数据以使占用连续扇区簇。该API包括一组文件系统控制：FSCTL_GET_VOLUME_BITMAP可以获得一个卷的空闲的扇区簇和已占用的扇区簇的情况；FSCTL_GET_RETRIEVAL_POINTERS可以获得一个文件扇区簇的情况；FSCTL_MOVE_FILE可以用于移动文件数据。

5.7 NTFS文件系统驱动程序

Win32 I/O API是通过I/O管理器来完成的。I/O管理器将I/O请求送交NTFS FSD去执行。在执行过程中，I/O管理器还要与高速缓存管理器、内存管理器、文件日志服务、卷管理器、磁盘驱动程序等一起协同完成I/O操作。参见图5-20。

应用程序通过NTFS 的FSD创建和存取文件。这一过程比较复杂，简要说来包括以下几个步骤：首先Windows 2000/XP进行有关使用权限的检查，只有合法用户的请求，才会被执行。然后I/O管理器将文件句柄转换为文件对象指针。最后NTFS通过文件对象指针来获得磁盘上的文件。

下面我们再来分析一下NTFS是如何通过文件对象指针来获得磁盘上的文件的。NTFS通过文件对象指针获得文件属性的流控制块(System Control Block, SCB)。每个SCB表示了文件的单个属性，并包含如何获得该属性的信息。同一个文件的所有SCB都指向一个共同的数据结构文件控制块(File Control Block, FCB)。FCB包含一个指向主文件表(Master File Table, MFT)中该文件记录的指针。NTFS通过该指针获得文件访问权。参见图5-21。

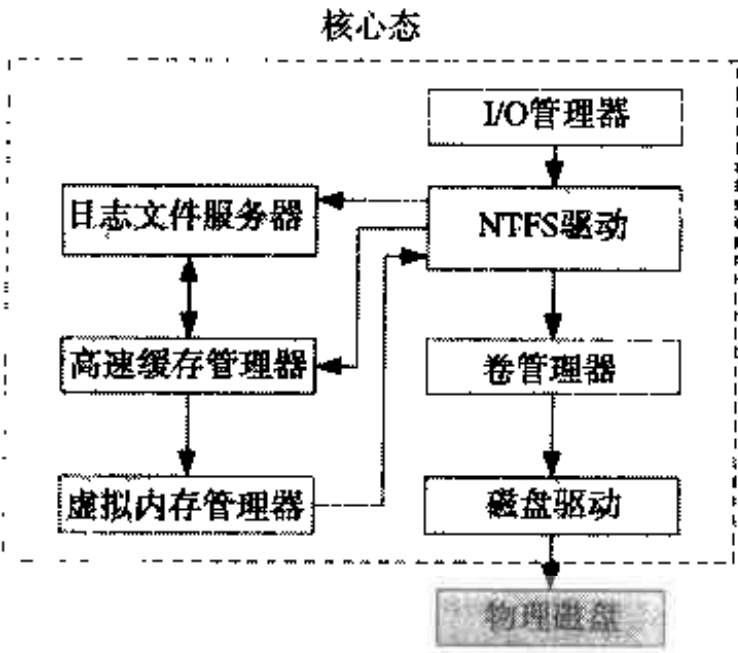


图5-20 NTFS及其相关组件图

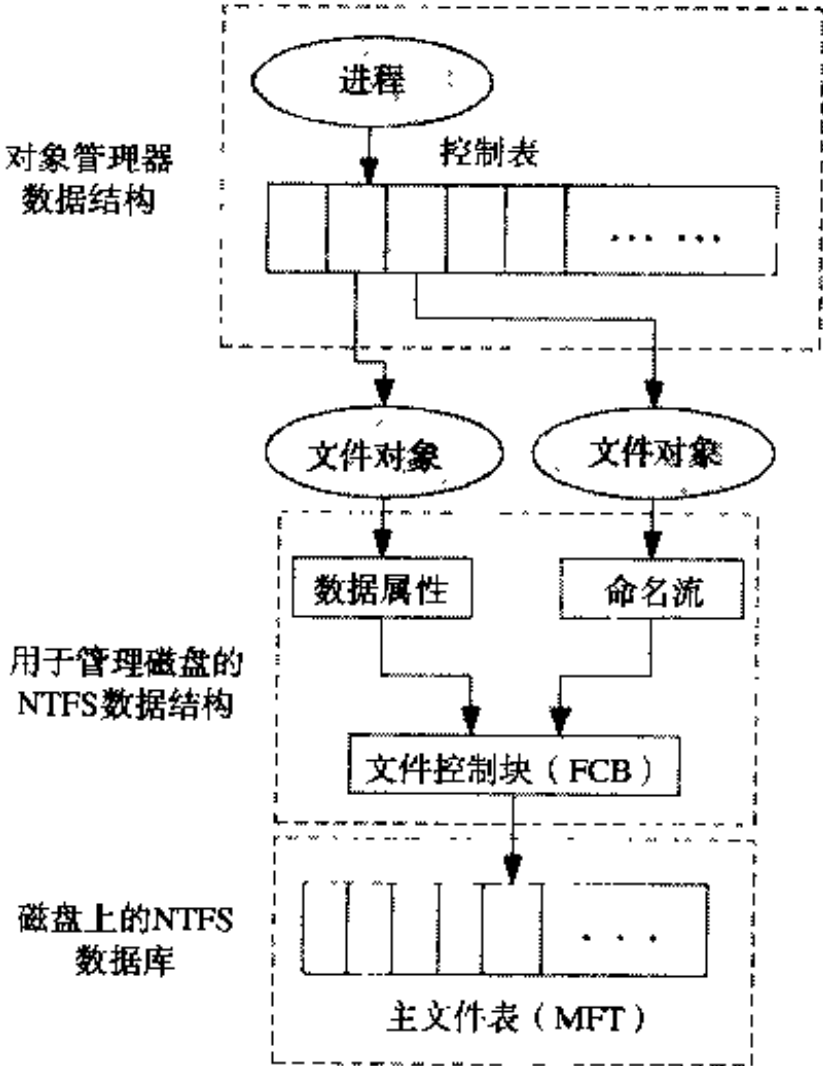


图5-21 NTFS数据结构

5.8 NTFS磁盘结构

如前所述，NTFS具有众多优点，这主要是因为它在磁盘上独特的实现方法。在本节中，我们主要针对NTFS来描述如何划分磁盘，如何组织文件与目录，如何存储文件属性与数据，以及如何压缩文件数据等。

5.8.1 卷

NTFS是以卷为基础的，而卷是建立在磁盘分区上。当以NTFS格式来格式化磁盘分区时就创建了NTFS卷。分区（partition）包括基本分区（primary partition）和扩展分区（extended partition）。扩展分区可以由逻辑分区（logical partition）组成。分区是磁盘的基本组成部分，是一个能够被格式化和单独使用的逻辑单元，在磁盘的自由空间上创建的（所谓自由空间，是指磁盘上没有被使用的或者说没有被分区的部分）。分区的目的主要有三个：一是使磁盘初始化，以便可以格式化和存储数据；二是用来分隔不同的操作系统，以保证多个操作系统在同一磁盘上正常运行；三是便于管理，可以有针对性地对数据进行分类存储，另外也可以更好地利用磁盘空间。

一个磁盘可以有多个卷，一个卷也可以由多个磁盘组成，如RAID磁盘阵列。Windows 2000/XP常使用两种卷：FAT卷和NTFS卷。图5-22显示了一个典型的磁盘分区情况。FAT卷不但包含所有存储的文件，而且还包含FAT文件系统所特有的区域，如FAT表。NTFS卷存储所有的文件系统数据，例如位图和目录，甚至包括作为一般文件的系统引导程序，这在以后将要详细介绍。

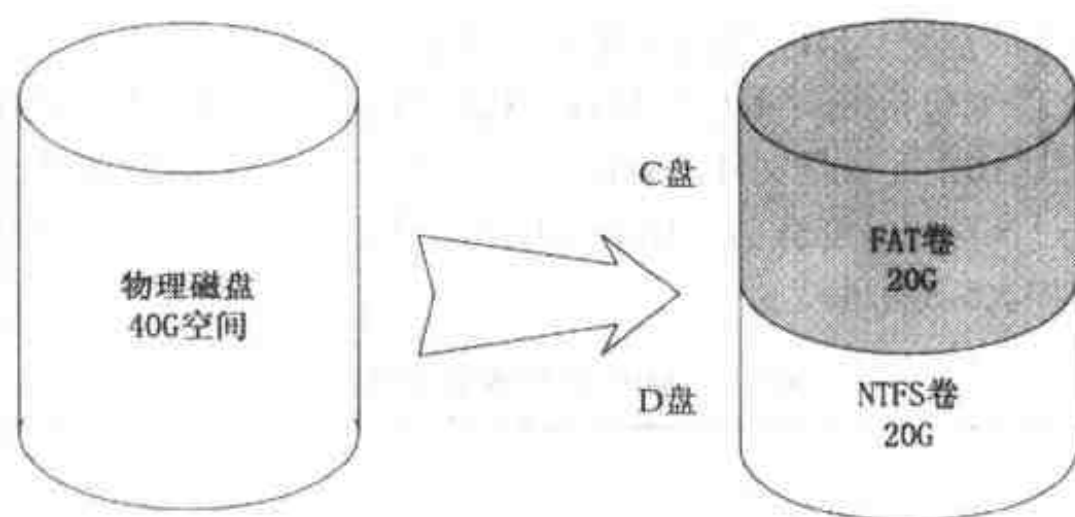


图5-22 典型的磁盘分区与卷

5.8.2 簇

NTFS与其他的文件系统如FAT一样，也是以簇作为磁盘空间分配和回收的基本单位。文件系统的存储空间是以簇为单位进行划分和管理的，即一个文件总是占用若干个整簇，文件所使用的最后一簇的剩余空间就不再使用，而是浪费掉了。通过簇来间接管理磁盘，而并不需要知道磁盘扇区的大小，这样就使NTFS保持了与磁盘物理扇区大小的独立性，从而能够为不同大小的磁

盘选择合适的簇。

卷上簇的大小，又称卷因子，是用户在使用Format命令或其他的格式化程序格式化卷时确定的。簇太小了会出现磁盘碎片，簇太大又会浪费磁盘空间（内部碎片）。因此，簇因子随卷的大小而不同，但都是物理扇区的整数倍，通常是2的幂，例如，1个、2个、4个、8个扇区等，相应地簇大小一般为512字节、1 KB、2 KB或是4 KB。小磁盘（≤ 512 MB）中的默认簇大小是512字节；对于1 GB的磁盘，默认的簇大小是1 KB；1 GB至2 GB之间的磁盘则为2 KB；对大于2 GB的磁盘，默认的簇大小是4 KB。

NTFS使用逻辑簇号（Logical Cluster Number，LCN）和虚拟簇号（Virtual Cluster Number，VCN）来进行簇的定位。LCN是对整个卷中所有的簇从头到尾所进行的简单编号。卷因子乘以LCN，NTFS就能够取得卷上的物理字节偏移量，从而得到了物理磁盘地址。VCN则是对属于特定文件的簇从头到尾进行编号，以便于引用文件中的数据。VCN可以映射成LCN，所以不要求物理上连续。

5.8.3 主控文件表

在NTFS中，所有的数据，甚至所有的元数据（metadata）（包括用于文件定位和恢复的数据结构、引导程序数据以及记录整个卷的分配状态的位图等）都存储在文件中。这对于文件系统来说是很不寻常的。但是将所有数据都存放在文件中可使得文件系统很容易进行数据定位和处理，而且，每个单独的文件可以通过安全描述符来保护。另外，如果磁盘的一个部分发生损坏，NTFS可以重新定位元数据，从而防止磁盘访问失效。

主控文件表（Master File Table，MFT）是NTFS卷结构的核心，是NTFS中最重要的系统文件，它包含了卷中所有文件的信息。MFT是以文件记录数组来实现的，每个文件记录的大小都固定为1 KB。卷上的每个文件（包括MFT本身）都有一行MFT记录。MFT开始的16个元数据文件是保留的。在NTFS中只有这16个元数据文件占有固定的位置。每个这样的元数据文件都有一个以“\$”开头的文件名称，不过该符号是隐藏的。16个元数据文件之后则是普通的用户文件和目录。

有关MFT的结构如表5-3所示。

表5-3 MFT的元数据文件记录

0	\$Mft	MFT本身
1	\$MftMirr	MFT镜像
2	\$LogFile	日志文件
3	\$Volume	卷文件
4	\$AttrDef	属性定义表
5	\$\	根目录
6	\$Bitmap	位图文件
7	\$Boot	引导文件
8	\$BadClus	坏簇文件
9	\$Secure	安全文件
10	\$UpCase	大写文件

(续)

11	\$Extended metadata directory	扩展元数据目录
12		预留
13		预留
14		预留
15		预留
>15		其他用户文件和目录

每个MFT记录都对应着不同的文件,如果一个文件有很多属性或是分散成很多碎片,就很可能需要多个文件记录。这时,存放其文件记录位置的第一个记录就叫做“基文件记录”(base file record)。

下面简要描述一下MFT的元数据文件。

MFT中的第1个记录就是MFT自身。由于MFT文件本身的重要性,为了确保文件系统结构的可靠性,系统专门为它准备了一个镜像文件(\$MftMirr),也就是MFT中的第2个记录。

第3个记录是日志文件(\$LogFile)。该文件是NTFS为实现可恢复性和安全性而设计的。当系统运行时,NTFS就会在日志文件中记录所有影响NTFS卷结构的操作,包括文件的创建和改变目录结构的命令,例如复制,从而在系统失败时能够恢复NTFS卷。

第4个记录是卷文件(\$Volume),它包含了卷名、被格式化的卷的NTFS版本和一个标明该磁盘是否损坏的标志位(NTFS系统以此决定是否需要调用Chkdsk程序来进行修复)。

第5个记录是属性定义表(\$AttrDef, attribute definition table),其中存放了卷所支持的所有文件属性,并指出它们是否可以被索引和恢复等。

第6个记录是根目录(\),其中保存了存放于该卷根目录下所有文件和目录的索引。在访问了一个文件后,NTFS就保留该文件的MFT引用,第二次就能够直接进行对该文件的访问。

第7个记录是位图文件(\$Bitmap)。NTFS卷的分配状态都存放在位图文件中,其中每一位(bit)代表卷中的一簇,标识该簇是空闲的还是已被分配了的。

第8个记录是引导文件(\$Boot),它是另一个重要的系统文件,存放着Windows 2000/XP的引导程序代码。该文件必须位于特定的磁盘位置才能够正确地引导系统。该文件是在Format程序运行时创建的,这正体现了NTFS把磁盘上的所有事物都看成是文件的原则。这也意味着虽然该文件享受NTFS系统的各种安全保护,但还是可以通过普通的文件I/O操作来修改。

第9个记录是坏簇文件(\$BadClus),它记录了磁盘上该卷中所有的损坏的簇号,防止系统对其进行分配使用。

第10个记录是安全文件(\$Secure),它存储了整个卷的安全性描述符数据库。NTFS文件和目录都有各自的安全描述符,为了节省空间,NTFS将具有相同描述符的文件和目录存放在一个公共文件中。

第11个记录为大写文件(\$UpCase, upper case file),该文件包含一个大小写字符转换表。

第12个记录是扩展元数据目录(\$Extended metadata directory)。

从第13到第16个记录为预留,目前尚未使用。

MFT的前16个元数据文件是如此重要，为了防止数据的丢失，NTFS系统在该卷文件存储部分的正中央对它们进行了备份，参见图5-23。

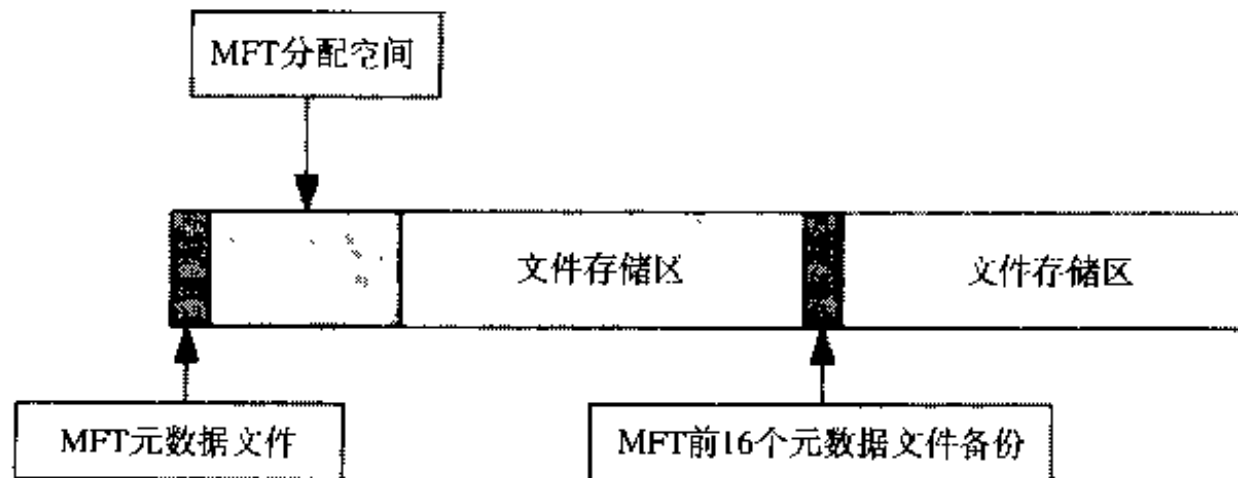


图5-23 MFT空间分配

NTFS把磁盘分成了两大部分，其中大约12%分配给了MFT，以满足其不断增长的文件数量。为了保持MFT元文件的连续性，MFT对这12%的空间享有独占权。余下的88%的空间被分配用来存储文件。而剩余磁盘空间则包含了所有的物理剩余空间——MFT剩余空间也包含在里面。MFT空间的使用机制可以这样来描述：当文件耗尽了存储空间时，Windows操作系统会简单地减少MFT空间，并把它分配给文件存储。当有剩余空间时，这些空间又会重新被划分给MFT。虽然系统尽力保持MFT空间的专用性，但是有时不得不做出牺牲。尽管MFT碎片有时是无法忍受的，我们却无法阻止它的发生。

那么NTFS到底是怎么通过MFT来访问卷的呢？首先，当NTFS访问某个卷时，它必须“装载”该卷：NTFS会查看引导文件（在图中的\$Boot元数据文件定义的文件），找到MFT的物理磁盘地址。然后它就从文件记录的数据属性中获得VCN到LCN的映射信息，并存储在内存中。这个映射信息定位了MFT的运行（run或extent）在磁盘上的位置。接着，NTFS再打开几个元数据文件的MFT记录，并打开这些文件。如有必要NTFS开始执行它的文件系统恢复操作。在NTFS打开了剩余的元数据文件后，用户就可以开始访问该卷了。

5.8.4 文件引用号

NTFS卷上的每个文件都有一个64位（bit）称为文件引用号（File Reference Number）的唯一标识。文件引用号由两部分组成：一是文件号，二是文件顺序号。文件号为48位（bit），对应于该文件在MFT中的位置。文件顺序号随着每次文件记录的重用而增加，这是为了让NTFS进行内部一致性检查而设计的。

5.8.5 文件记录

NTFS将文件作为属性/属性值的集合来处理，这一点与其他文件系统不一样。文件数据就是未命名属性的值，其他文件属性包括文件名、文件拥有者、文件时间标记等。图5-24显示了一个用于小文件的MFT记录。



图5-24 小文件的MFT记录

每个属性由单个的流(stream)组成,即简单的字符队列。严格地说,NTFS并不对文件进行操作,而只是对属性流的读写。NTFS提供对属性流的各种操作:创建、删除、读取(字节范围)以及写入(字节范围)。读写操作一般是针对文件的未命名属性的,对于已命名的属性则可以通过已命名的数据流句法来进行操作。

NTFS卷上文件的常用属性在表5-4中列出(并不是所有文件都有所有这些属性)。

表5-4 NTFS卷上常用属性说明

属 性 名	属 性 描 述
\$VOLUME_INFORMATION	卷信息:仅存于\$Volume元数据文件
\$VOLUME_NAME	卷名称或卷标识:仅存于\$Volume元数据文件
\$STANDARD_INFORMATION	标准信息:这包括基本文件属性,如只读、存档;时间标记,如文件的创建时间和最近一次修改的时间;有多少目录指向本文件(即它的硬链接数(hard link count))
\$FILE_NAME	文件名:这是以Unicode字符表示的,由于MS-DOS不能正确识别Win32子系统创建的文件名,当Win32子系统创建一个文件名时,NTFS会自动生成一个备用的MS-DOS文件名,所以一个文件可以由多种文件名属性
\$SECURITY_DESCRIPTOR	安全描述符:这是为了向后兼容而保留的,主要用于保护文件以防止未授权访问,但是,Windows 2000/XP已将所有文件的安全描述符存放在\$Secure元数据文件中,以便于共享(NTFS的早期版本将安全描述符与文件目录一起存放,这不利于共享)
\$DATA	文件数据:这是文件的内容(在NTFS文件系统中,一个文件除了支持文件数据即未命名的属性外,还可支持其他命名属性;目录没有默认的数据属性,但是有可选的命名数据属性)
\$INDEX_ROOT	索引根
\$INDEX_ALLOCATION	索引分配
\$BITMAP	位图
\$ATTRIBUTE_LIST	属性列表:当一个文件需要使用多个MFT文件记录时,这用来表示该文件的属性列表,因此,不常使用
\$OBJECT_ID	对象ID:一个具有64个字节的标识符,其中最低的16个字节对卷来说是唯一的(链接跟踪服务为外壳快捷及OLE链接源文件赋予对象ID;NTFS提供API来直接通过这些对象ID而不是文件名来打开文件)
\$REPARSE_POINT	重解析点:存储文件的重解析点数据(NTFS的软链接与装配点都包括这个属性)
\$EA	扩充属性:主要为与OS/2兼容,现已使用不多
\$EA_INFORMATION	扩充属性信息:主要为与OS/2兼容,现已使用不多
\$LOGGED_UTILITY_STREAM	EFS加密属性:主要为实现EFS(encrypted file system)而存储有关加密信息如解码密钥、合法访问的用户列表等

5.8.6 文件名称

NTFS和FAT路径名中的每个文件名/目录名的长度可达255个字节，可以包含Unicode字符、多个空格及句点。但是，MS-DOS文件系统只支持8个字符的文件名加上3个字符的扩展名。由于MS-DOS不能正确识别Win32子系统创建的文件名，当Win32子系统创建一个文件名时，NTFS会自动生成一个备用的MS-DOS文件名。

POSIX子系统则需要Windows 2000/XP支持的所有应用程序执行环境中的最大的名字空间，因此NTFS的名字空间等于POSIX的名字空间。POSIX子系统甚至可以创建在Win32和MS-DOS中不可见的名称。

图5-25说明了MS-DOS子系统、Win32子系统和POSIX子系统所支持的名字空间之间的关系。

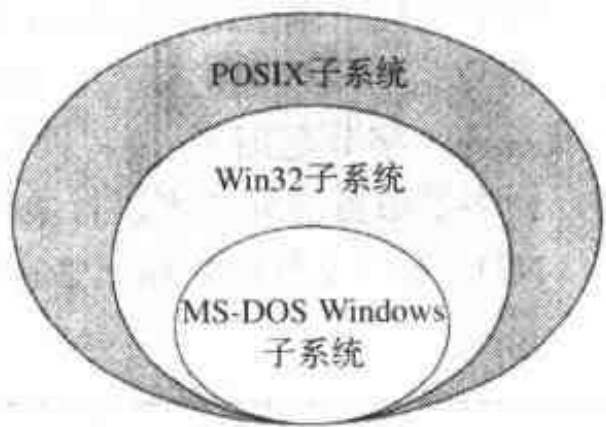


图5-25 MS-DOS子系统、Win32子系统和POSIX子系统的名字空间

5.8.7 常驻属性与非常驻属性

当一个文件很小时，其所有属性和属性值可存放在MFT的文件记录中。当属性值能直接存放在MFT中时，该属性就称为常驻属性（resident attribute）。有些属性总是常驻的，这样NTFS才可以确定其他非常驻属性。例如，标准信息属性和根索引就总是常驻属性。

每个属性都是以一个标准头开始的，在头中包含该属性的信息和NTFS通常用来管理属性的信息。该头总是常驻的，并记录着属性值是否常驻。对于常驻属性，头中还包含着属性值的偏移量和属性值的长度。

如果属性值能直接存放在MFT中，那么NTFS对它的访问时间就将大大缩短。NTFS只需访问磁盘一次，就可立即获得数据；而不必像FAT文件系统那样，先在FAT表中查找文件，再读出连续分配的单元，最后找到文件的数据。

小文件或小目录的所有属性，均可以在MFT中常驻。小文件的未命名属性可以包括所有文件数据。小目录的索引根属性可以包括其中所有文件和子目录的索引。参见图5-26。



图5-26 小目录的MFT记录

大文件或大目录的所有属性，就不可能都常驻在MFT中。如果一个属性（如文件数据属性）太大而不能存放在只有1 KB的MFT文件记录中，那么NTFS将从MFT之外分配区域。这些区域通常称为一个运行（run）或一个盘区（extent），它们可用来存储属性值，如文件数据。如果以后属性值又增加，那么NTFS将会再分配一个运行，以使用来存储额外的数据。值存储在运行中而不是在MFT文件记录中的属性称为非常驻属性（nonresident attribute）。NTFS决定了一个属性是常驻还是非常驻的；而属性值的位置对访问它的进程而言是透明的。

当一个属性为非常驻时，如大文件的数据，它的头包含了NTFS需要在磁盘上定位该属性值的有关信息。图5-27显示了一个存储在两个运行中的非常驻属性。

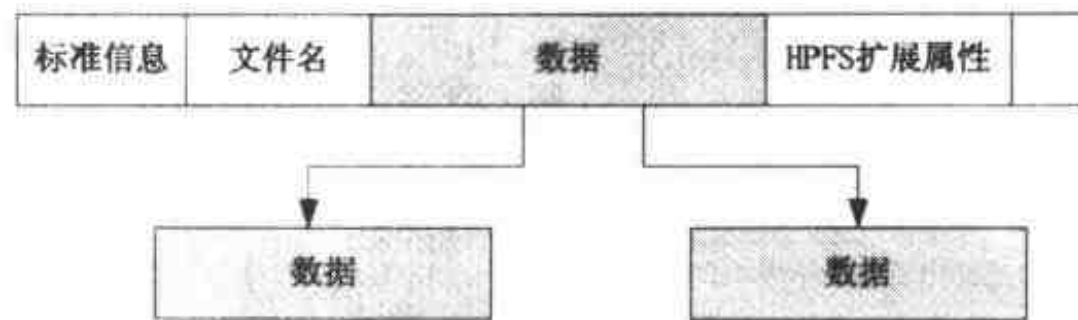


图5-27 存储在两个运行中的非常驻属性

在标准属性中，只有可以增长的属性才是非常驻的。对文件来说，可增长的属性有数据、属性列表等。标准信息 and 文件名属性总是常驻的。

一个大目录也可能包括非常驻属性（或属性部分），参见图5-28。在该例中，MFT文件记录没有足够空间来存储大目录的文件索引。其中，一部分索引存放在索引根属性中，而另一部分则存放在叫作“索引缓冲区”（index buffer）的非常驻运行中。这里，索引根、索引分配以及位图属性都是简化表示的，这些属性将在后面详细介绍。对目录而言，索引根的头及部分值应是常驻的。

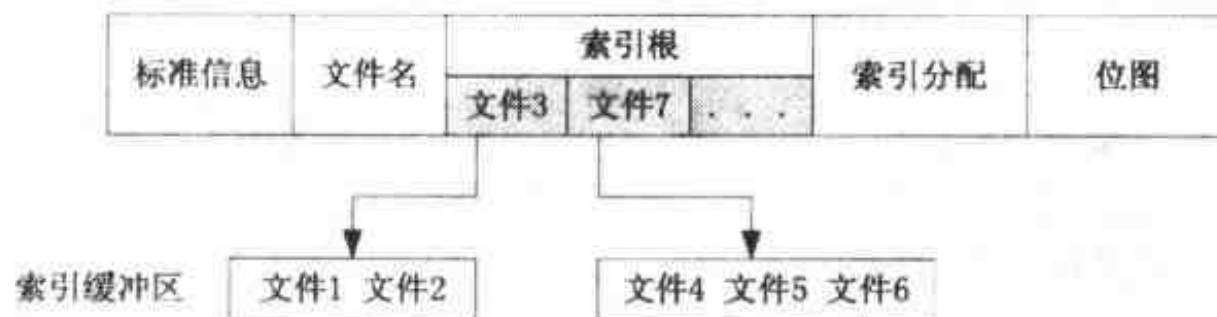


图5-28 大目录的MFT记录

当一个文件（或目录）的属性不能放在一个MFT文件记录中，而需要分开分配时，NTFS通过VCN-LCN之间的映射关系来记录运行（run）或盘区情况。LCN用来为整个卷中的簇按顺序从0到n进行编号，而VCN则用来对特定文件所用的簇按逻辑顺序从0到m进行编号。图5-29显示了一个非常驻数据属性的运行所使用的VCN与LCN编号。

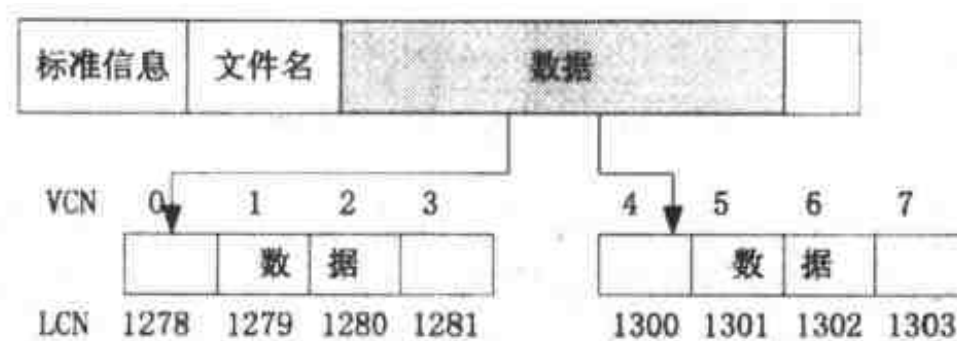


图5-29 非常驻数据属性的VCN

当该文件含有超过2个运行时，则第三个运行从VCN 8开始，数据属性头部含有前两个运行VCN映射，这便于NTFS对磁盘文件分配的查询。为了便于NTFS快速查找，具有多个运行文件的常驻数据属性头中包含了VCN-LCN映射关系，参见图5-30。

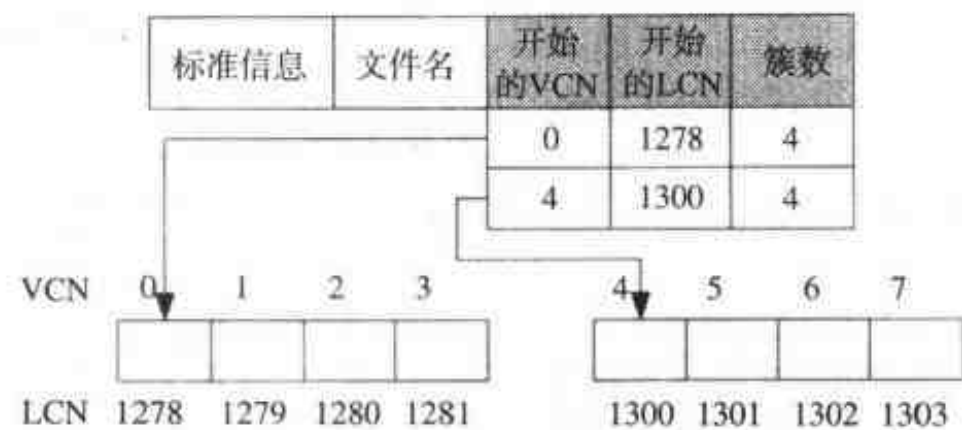


图5-30 非常驻数据属性的VCN-LCN映射

虽然数据属性常常因太大而存储在运行中，但是其他属性也可能因MFT文件记录没有足够空间而需要存储在运行中。另外，如果一个文件有太多的属性而不能存放在MFT记录中，那么第二个MFT文件记录就可用来容纳这些额外的属性（或非常驻属性的头）。在这种情况下，一个叫作“属性列表”（attribute list）的属性就加进来。属性列表包括文件属性的名称和类型代码以及属性所在MFT的文件引用。属性列表通常用于太大或太零散的文件，这种文件因VCN-LCN映射关系太大而需要多个MFT文件记录。具有超过200个运行的文件通常需要属性列表。

5.8.8 索引

在NTFS系统中，文件目录仅仅是文件名的一个索引。NTFS使用了一种特殊的方式把文件名组织起来，以便于快速访问。当创建一个目录时，NTFS必须对目录中的文件名属性进行索引。图5-31显示了一个卷的MFT的根目录记录。

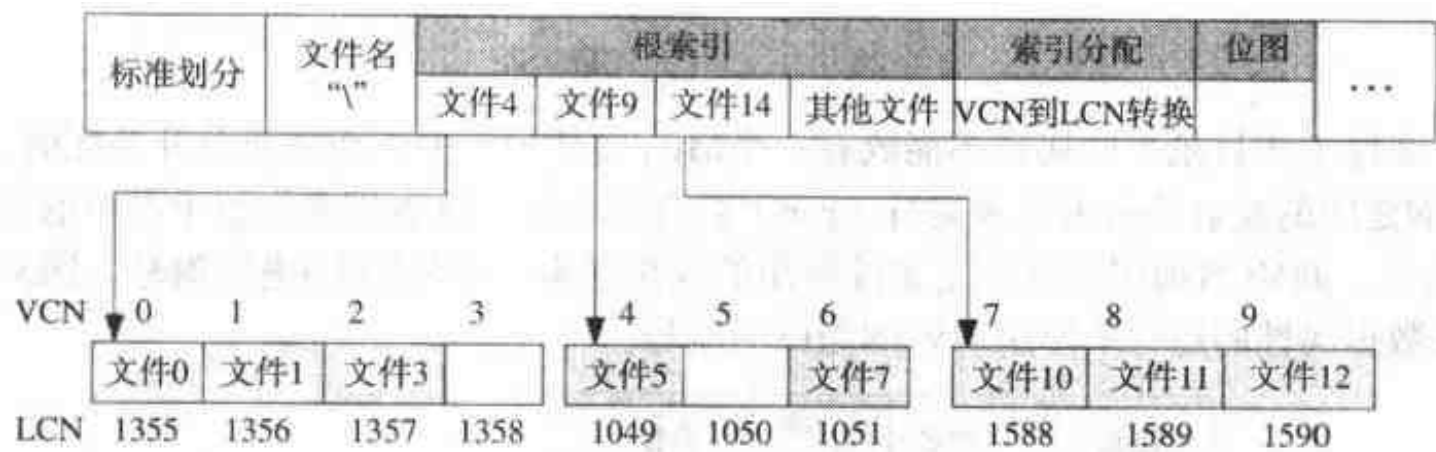


图5-31 根目录的文件索引

一个目录的MFT记录将其目录中的文件名和子目录名进行排序，并保存在索引根属性中。然而，对于一个大目录，文件名实际存储在组织文件名的固定4 KB大小的索引缓冲区中。索引缓冲区是通过B+树数据结构实现的。B+树是平衡树的一种，对于存储在磁盘上的数据来说，平衡树是一种理想的分类组织形式，因此使查找一个项时所需的磁盘访问次数减到最少。根索引属性包含B+树的第一级（根子目录）并指向包含下一级（大多数是子目录，也可能是文件）的索引缓冲区中。

上图只显示了根索引属性中的文件名和索引缓冲区。但是索引中的每一项还包括了位于MFT

中的描述文件所在位置的文件引用以及文件时间和文件大小等信息。NTFS根据文件的MFT记录来复制时间标记和文件大小信息。这种技术需要将更新信息写在两个地方，因此比较麻烦。但是，这仍是一个提高目录浏览速度的好方法，因为它可以在文件系统不打开目录中任何文件的情况下显示每个文件的时间标记和大小。

索引分配属性包含了索引缓冲区的VCN到LCN映射，而位图属性跟踪在索引缓冲区中哪些VCN是在使用而哪些是空闲的。上图中显示了每个文件项占有一个VCN，而实际上多个文件项被包装在同一个簇中。每个4 KB大小的索引缓冲区可以容纳20到30个文件项。

5.8.9 数据压缩

数据压缩是NTFS文件系统的一个重要特征。虽然FAT文件系统也支持数据压缩，但是NTFS压缩功能可以对单个文件、整个目录或NTFS卷上的整个目录树进行压缩（NTFS压缩只在用户数据上进行，而不能在文件系统元数据上进行）。

数据压缩可以减少磁盘使用空间，但是由于每次解压缩需要大量的数据运算，使用压缩功能将会导致NTFS卷的性能下降。如果要拷贝一个压缩文件，其过程是：解压缩、拷贝、重新对拷贝的文件进行压缩，这些都大大增加了CPU的处理时间。

下面，将分别介绍对稀疏文件和非稀疏文件的解压缩。

1. 压缩稀疏文件

稀疏文件是那些只包含少量非零数据和大量零数据的文件，例如磁盘上的稀疏矩阵。这种文件一般具有很大的压缩潜力，通常可以压缩成只占原来磁盘的一小部分。它的压缩方式也十分简单：NTFS只给那些包含非零数据的运行分配磁盘空间，参见图5-32。

标准信息	文件名	开始的VCN	开始的VCN	簇数
		0	1200	16
		32	1280	16
		64	1356	16
		80	967	16

图5-32 稀疏文件压缩的MFT记录

当程序从压缩文件中读取数据时，NTFS通过检测该位置是否有VCN到LCN的映射来决定该数据是不是零数据。若有映射，则为非零数据，需要从磁盘上读取；若没有映射，即存在尚未分配的“空洞”（unallocated hole），则为零数据，就直接返回零数据。当然，如果程序要把非零数据写到空洞中时，系统需要先分配空间，再写入数据。

2. 压缩非稀疏文件

NTFS是以16个簇为压缩单元来进行一般文件的压缩的。NTFS要决定这16个单元压缩后是否能至少腾出1个单元来。如果能够节省存储空间，那么NTFS就只分配容纳压缩所需的簇数并将数

据写到磁盘上；否则，NTFS系统就放弃这种努力，直接将数据写到磁盘上。当NTFS向压缩文件写数据时，它确保每个运行都以一个虚拟16簇边界开始。因此每个运行中VCN都是以16的倍数开始的，并且运行的长度不大于16。选用16簇作为压缩的单元是为了减少内部碎片，是减小压缩文件和降低随机访问文件程序读取操作之间的平衡（因为读取每个压缩单元都要进行复杂的解压缩运算）。这种压缩方式同时也非常有弹性，甚至有时会产生很有趣的情况：一个文件可能前半部分被压缩而后半部分基本保持原样，如图5-33所示。

标准信息	文件名	开始的VCN	开始的LCN	簇数
		0	1200	4
		16	1280	8
		32	1356	9
		48	967	16

图5-33 非稀疏文件压缩的MFT记录

为了提高读取压缩文件的速度，NTFS先把数据解压缩到高速缓存缓冲区中，然后再复制到调用缓冲区中。同时NTFS也将解压缩的数据加载到高速缓存中，从而使后续的从同一个运行上进行的读取操作在任何其他的高速缓存上一样快。当两个运行在磁盘上相邻时，NTFS也执行磁盘预读。同样，在写文件的过程中，NTFS在高速缓存中更新文件后，由延迟写线程异步进行压缩，并将修改后的数据写到磁盘上。这些做法都大大提高了压缩文件的读写性能。

5.9 NTFS可恢复性支持

NTFS通过日志记录（logging）来实现文件系统的可恢复性。所有改变文件系统的子操作在磁盘上运行以前，首先被记录在日志文件中。在系统崩溃后的恢复阶段，NTFS根据记录在日志文件中的文件操作信息，对那些部分完成的事务进行重做或是撤销，从而保证了磁盘上文件系统的一致性。这种技术称为“预写日志记录”（write-ahead logging）。

5.9.1 日志记录的实现

下面我们来讨论日志记录是如何在NTFS上实现的。

1. 日志文件服务

日志文件服务（Log File Service, LFS）是一组NTFS驱动程序内的核心态程序。NTFS是通过LFS例程来访问日志文件的。它的结构参见图5-34。

LFS将日志文件分为两个区域：重启动区域（restart area）和无限记录区域（infinite logging area）。它的结构参见图5-35。

NTFS使用重启动区域来存储相关信息，在系统失败后的恢复过程中，NTFS将从这个位置开始读取信息。由于重启动区域的重要性，在紧随其后的磁盘空间上，LFS保存了它的一个副本。

在LFS重新启动区域之后，为记录区域，用于存放NTFS的日志记录

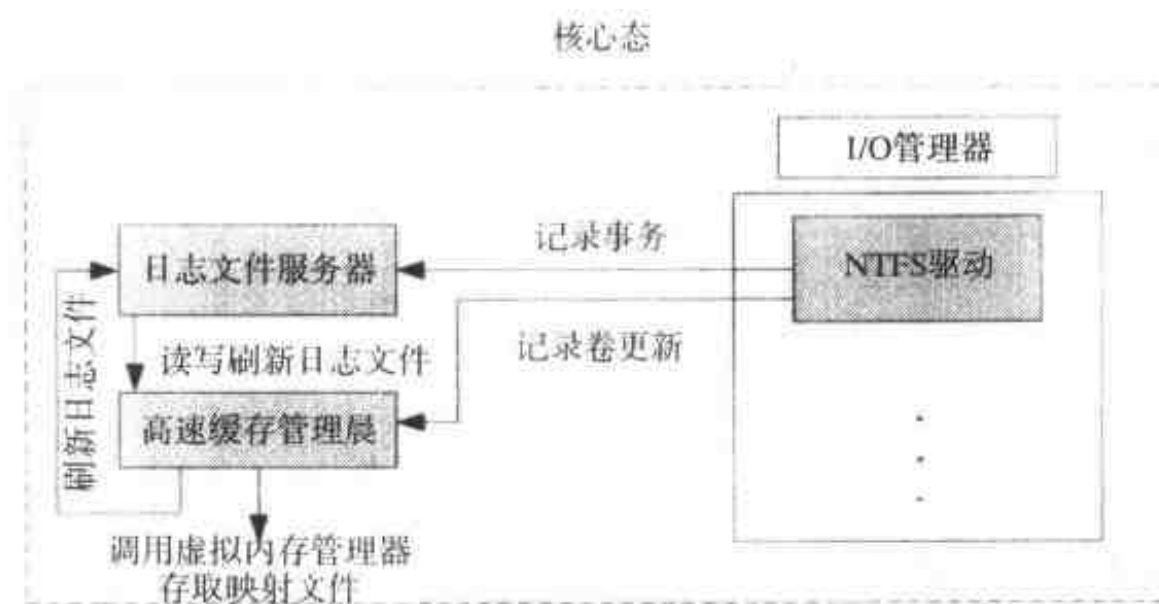


图5-34 LFS结构示意图



图5-35 日志文件结构

LFS利用逻辑序列号（Logic sequence numbers, LSN）来标识日志文件中的记录。LFS为64位(bit)。通过循环使用日志记录，LFS使日志文件看起来像是可以保存无限多的日志记录。

NTFS不会直接从日志文件中读写记录，而是通过LFS来读写记录的。LFS提供了许多操作来处理日志文件，包括打开（open）、写入(write)、向前(prev)、向后（next）、更新（update）等操作。在恢复过程中，NTFS通过向前读取日志记录，重做已在日志文件中记录的、系统失败时还没有及时刷新到磁盘上的所有事务；NTFS通过向后读取日志记录，撤消或是回退系统崩溃前没有完全记录在日志文件中的事务。当NTFS不再需要日志文件中较早的事务记录时，它就调用LFS来将日志文件的开始部分设置为一个具有更高LSN的记录，从而实现日志记录的“无限”使用。

下面是NTFS为实现卷的可恢复性而执行的操作步骤：

- 1) NTFS首先调用LFS在日志文件中记录所有改变卷结构的事务。
- 2) NTFS执行在高速缓存中的更改卷结构的操作。
- 3) 高速缓存管理器调用LFS将日志文件刷新到磁盘。
- 4) 完成上一步之后，卷更改（事务本身）最后被刷新到磁盘上。

严格执行这些操作步骤就保证了即使文件系统的最终修改是不成功的，通过日志文件也能恢复相应的事务。重新引导系统以后，当第一次使用卷时，文件系统的恢复工作就自动开始。如此就保证了无论何时发生意外，NTFS都可以通过日志文件记录中的操作信息来恢复文件系统的一致性。

2. 日志记录类型

LFS允许用户在日志文件中写入任何类型的记录。更新记录（update records）和检查点记录（checkpoint record）是NTFS所支持的两种主要类型的日志记录。它们在系统的恢复过程中起了

主要作用。下面作一简要介绍。

(1) 更新记录

更新记录所记录的是文件系统的更新信息，是NTFS写入日志文件中的最普通的记录类型。NTFS为以下事务写入更新记录。

- 创建文件
- 删除文件
- 扩展文件
- 截断文件
- 设置文件信息
- 重新命名文件
- 更改应用于文件的安全信息

在更新记录中一般包含两种信息：

- **重做信息** 如果事务在高速缓存中的操作记录刷新到磁盘之前系统崩溃，如何重新执行这个对卷来说是已提交的事务子操作。
- **撤消信息** 当系统失败时，如何撤消这个对卷来说未提交的事务子操作。

创建一个新文件事务的日志文件记录结构如图5-36所示。

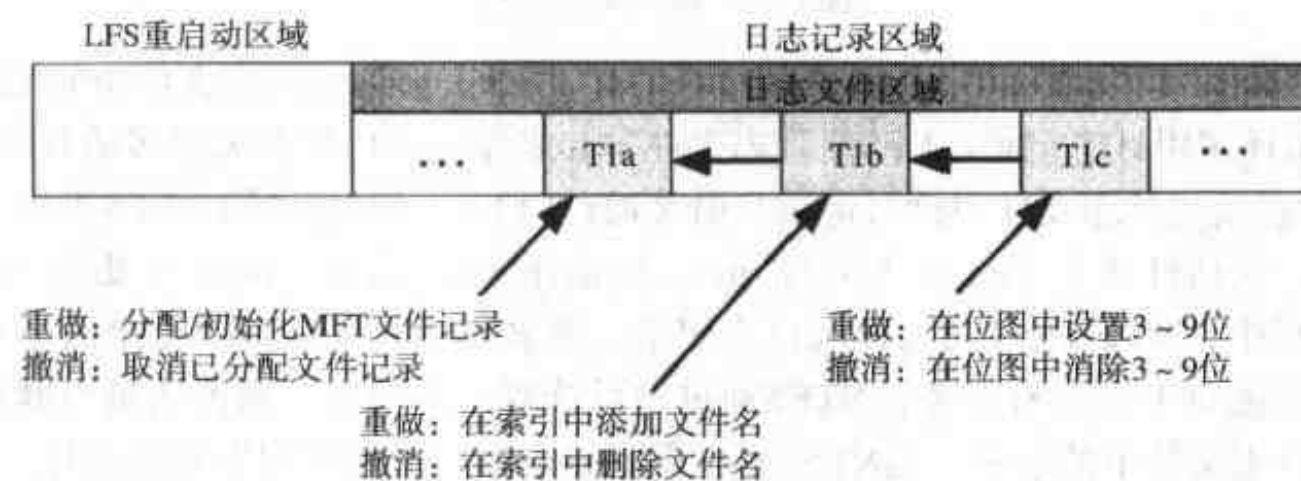


图5-36 日志文件中的更新记录

其中每个记录代表了该事务的一个子操作。NTFS根据每个更新记录中的重做项来决定如何重新执行该子操作，而根据撤消项来决定如何执行回退子操作。

当一个事务的最后一个子操作被记录后，NTFS就对高速缓存中的卷自身执行子操作。在完成高速缓存的更新以后，NTFS就向日志文件写入该事务的最终记录——被称为“提交一个事务”的子操作，完整地记录整个事务，从而完成了该事务的提交。这时，即使操作系统立即发生崩溃，NTFS也能保证卷上该事务的完整性。

当发生系统失败需要进行恢复时，NTFS根据读取的日志文件信息，重做每一个已经提交的事务。由于NTFS并不清楚已经提交的事务是否从高速缓存中得到及时更新，所以如果在事务最终提交以前发生系统崩溃，NTFS就再一次执行已经提交的事务，从而保持磁盘的一致性。

在文件系统恢复过程中完成了重做操作之后，NTFS根据系统崩溃时未被提交的事务日志文

件中的撤消信息来回退已经记录的每一个子操作。在上图中，NTFS首先撤消了T1c子操作，然后是T1b，依次类推，直到事务中的第一个子操作。

对于重做和撤消信息我们可以采用物理或是逻辑的表达方式。物理表达是根据磁盘上特定的字节范围来指定卷的更新，这些字节可以被更改、移动等等；而逻辑表达则是根据操作来表达更新信息，例如，“删除文件1”等等。当在软件的最低层维护文件系统结构时，NTFS根据物理表达写入更新记录。事务处理或其他应用程序级软件则可能得益于用逻辑表达来写入更新记录，因为逻辑表达的更新比物理表达的更新更加简洁。

NTFS设计小组对更新记录中的重做和撤消信息结构进行了仔细而慎重的设计，使其保持信息的完备性。防止NTFS试图重做一个已经做过的事务，或相反地，试图撤消一个根本没有进行或是已经被撤消的事务。类似地，NTFS也可能试图重做在磁盘上只是部分完成的几个更新记录组成的事务。更新记录的格式必须保证执行冗余重做或是撤消操作是“幂等”（idempotent）的，也就是说，具有中立的作用。例如，设置一个已经设置的位不会起作用（可以冗余重做），但是切换一个已经切换的位，就会起作用（不可以冗余重做）。

(2) 检查点记录

除了更新记录之外，NTFS还周期性地向日志文件中写入检查点记录。NTFS在写入检查点记录以后，还在重启动区域存储记录的LSN。在发生系统失败后的恢复过程中，NTFS通过存储在检查点记录中的信息来定位日志文件中的恢复点，参见图5-37。



图5-37 日志文件中的检查点记录

随着日志记录的增长，虽然NTFS可以不断检查并释放日志文件的空间，但是日志文件也还是有可能被填满的。LFS通过跟踪以下数值来做出判断：

- 可用的磁盘空间。
- 在日志文件中写入一个新的日志记录和撤消该写入所必须的空间大小。
- 回退所有未提交事务所必需的空间大小。

如果最后两项所需空间的总和超过了日志文件的可用空间，LFS将返回一个“日志文件已满”的错误，并且引起一个NTFS异常。NTFS异常处理程序将回退到当前事务，并将其放置在一个队列中，以便在调整空间以后重新启动它。

为了释放日志文件中的空间，NTFS必须暂时防止进一步的事务：NTFS首先停止文件的创建和删除，然后请求获得对所有系统文件的独占访问和对所有用户文件的共享访问。这样逐渐地，活动事务或者成功完成，或者因日志文件已满而引起异常。对于异常，NTFS将回退到当前事务，并将其放置在队列中，以便重新启动后再用。

一旦NTFS开始释放日志文件的空间，它将调用高速缓存管理器，将所有未写入的数据，包括未写入的日志文件数据，都刷新到磁盘上。在NTFS完成每个事务的安全刷新后，它就清空失去作用的日志文件，把当前位置重新设置为日志文件的开始部分。然后NTFS重新启动已排队的事务。

5.9.2 可恢复性实现

NTFS通过LFS来实现可恢复功能。NTFS的可恢复性支持确保了系统发生意外时磁盘卷结果的完整性和一致性。即便是很大的磁盘，也能在几秒钟之内恢复过来。需要注意的是，这种恢复只是针对文件系统的数据，而并不能保证用户数据完全被恢复。

NTFS在内存中维护两张表：

- 事务表：跟踪已经启动但尚未提交的事务。在恢复过程中，必须从磁盘删除这些活动事务的子操作。
- 脏页表：记录了在高速缓存中还未写入磁盘的包含改变NTFS卷结构操作的页面。在恢复过程中，这些改动必须刷新到磁盘上。

NTFS每隔5秒钟向日志文件写入一个检查点记录。在此之前，NTFS调用LFS在日志文件中存储事务表和脏页表的一个当前副本。这样，NTFS写入的检查点记录就包含了已复制表的日志记录的LSN。当系统失败后开始恢复时，NTFS调用LFS来定位日志文件记录，这些日志记录包含了最近的检查点记录以及最近的事务表和脏页表的副本。然后，NTFS将这些表复制到内存。

在最近的检查点记录之后，日志文件通常包含更多的更新记录。这些更新记录显示了在最后的检查点记录写入后卷的更改。为此，NTFS必须更新事务表和脏页表。通过更新这些表和日志文件中的内容来更新卷本身。

要实现NTFS卷的恢复，NTFS要对日志文件进行三次扫描：

- 分析扫描 (analysis pass)
- 重做扫描 (redoing pass)
- 撤消扫描 (undoing pass)

1. 分析扫描

NTFS从日志文件中最近的一个检查点操作的起点开始分析扫描。检查点操作起点之后的每一个更新记录都代表对事务表或脏页表的修改，如“事务提交”记录代表的事务必须从事务表中删除，“页面更新”记录则表示因为对一个文件系统数据结构作了修改，相应的脏页表也必须更新。参见图5-38。

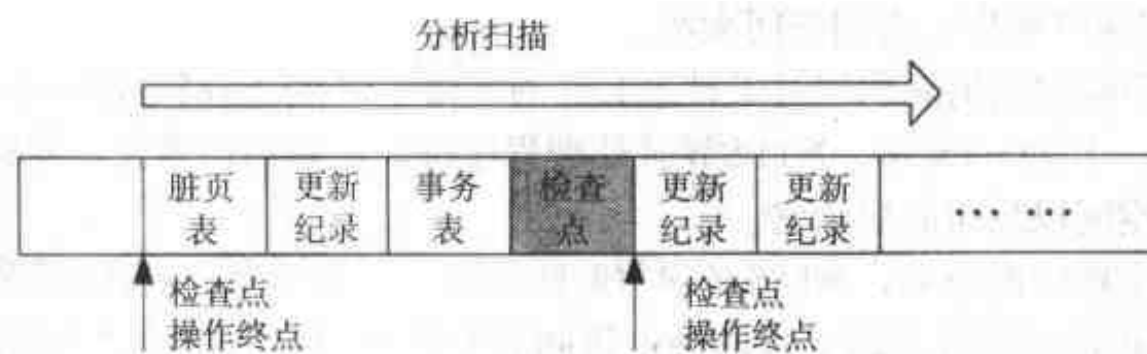


图5-38 分析扫描

这两个表被复制到内存中以后，NTFS将搜索这两个表。事务表包含了未提交（不完整）事务的LSN，脏页表则包含了高速缓存中还未刷新到磁盘的记录的LSN。NTFS根据其中的信息来确定最早的更新记录（该记录记录了在磁盘上尚未进行的操作）的LSN，由此决定重做扫描的起

点。当然，如果最近一个检查点记录更早，NTFS将从那里开始启动重做扫描。由此进入事务恢复的第二阶段。

2. 重做扫描

在重做扫描过程（参见图5-39）中，NTFS将从分析扫描得到的最早记录的LSN开始，在日志文件中向前扫描。NTFS将查找“页面更新”记录，这个记录包含了在系统失败前就已经写入的卷更新，但是这些卷的更改可能还未刷新到磁盘。NTFS将在高速缓存中重做这些更新。当NTFS到达日志文件的末端时，它已经利用必要的卷更改更新了高速缓存，高速缓存管理器的延迟写线程能够开始在后台向磁盘写入高速缓存的内容。

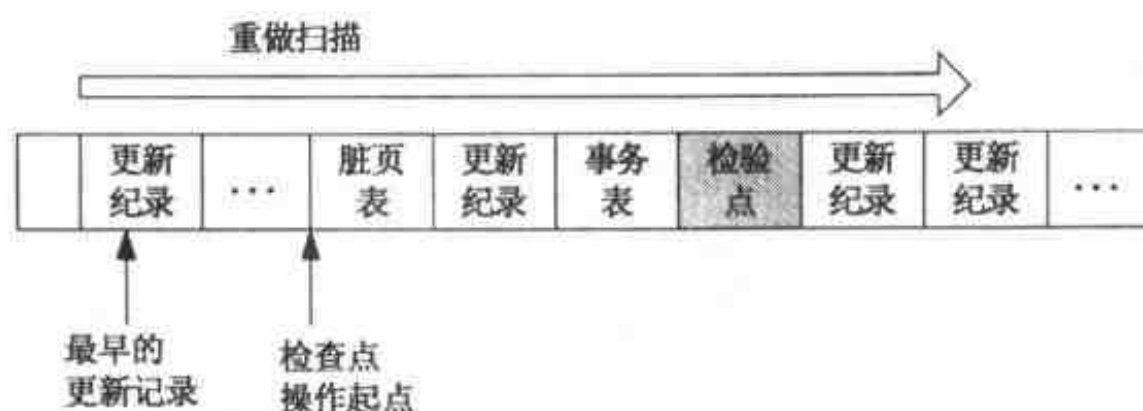


图5-39 重做扫描

3. 撤消扫描

在NTFS完成重做扫描后，它将开始撤消扫描（图5-40）。NTFS可以在这一扫描中回退系统失败时任何未提交的事务。

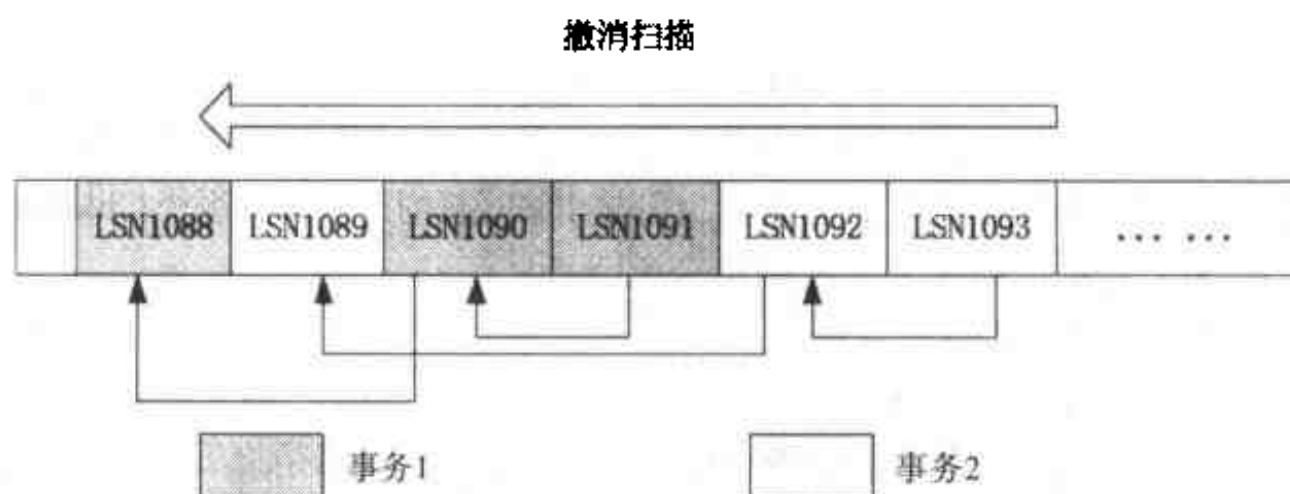


图5-40 撤消扫描

上图中有两个事务：事务1在断电时已经提交，事务2未提交。假设事务2创建一个文件，有3个子操作，它们之间通过指针链接。事务表为每个未提交的事务更新记录列出最后记录的LSN。

每个更新记录包含两种信息，一个是如何重做一个子操作，另一个是如何撤消子操作。NTFS在定位LSN1093后，执行撤消操作直到LSN1089，完成事务2的回退。当然，在日志文件中也要记录撤消操作，因为撤消时也可能发生系统崩溃。

恢复完成后，NTFS将高速缓存写入磁盘从而保证卷是最新的。最后，NTFS写入一个“空”到LFS重启动区，指明卷是一致的。这时，即使系统再次崩溃，也不必再恢复了。

由于NTFS使用的是“延迟提交”的算法，这意味着每次“事务提交”记录写入时，日志文件不能立即刷新到磁盘，而是被批处理写入的。同时，多个事务可能是平行操作的，它们的事务提交记录可能一部分被写入磁盘，而另一部分没有写入。这样就只能保证NTFS恢复到某一先前存在的一致状态，而不能保证NTFS恢复到刚巧系统崩溃时的状态。

NTFS还能够利用日志记录实现文件系统错误的恢复。因为NTFS日志记录了每个更改卷结构的事务，包括正常文件I/O过程中发生的文件系统错误，所以日志文件可大大简化文件系统的错误处理代码。当然，一个程序收到的大多数I/O错误不是文件系统错误，调用者必须适当地依次响应错误。

5.10 NTFS坏簇恢复支持

NTFS通过卷管理工具和容错磁盘驱动支持，增加了磁盘的数据冗余和容错功能，为文件系统数据提供了极高的可靠性。尽管即使没有卷管理工具，NTFS也可以从使用中删除坏簇并提供非SCSI磁盘的坏扇区恢复功能。如果再加上FtDisk.exe的配合，NTFS可以提供最高级的数据完整性。

关于NTFS对RAID的支持，前面已做了深入讨论，因而这里就只讨论一下NTFS对坏簇的恢复。

Windows 2000/XP卷管理功能分别通过用于基本磁盘的FtDisk和用于动态磁盘的LDM (Logical Disk Manager) 的卷管理工具来实现坏簇的修复。NTFS在系统运行时动态收集有关坏簇的资料，并把这些资料储存在系统文件里。这样，NTFS就隐藏了坏簇恢复的细节，以至于在应用程序环境里根本不必知道坏簇的存在。

如果一个扇区发生错误并且磁盘不能提供备用扇区，NTFS卷管理工具程序会给系统发出警告。当卷管理器返回一个坏扇区警告，或是当磁盘驱动程序返回坏扇区错误信息时，NTFS分配一个新的簇替换包含坏扇区的簇。NTFS动态地替换包含坏扇区的簇，并且跟踪这些簇，以保证它们不被重新使用。FAT文件系统并不响应这个警告，必须由用户自己运行Chkdsk或是Format进行坏簇恢复，然而事实证明这种修复方法很难令人满意。

NTFS通过和卷管理工具的配合最大限度地减少了坏簇对整个文件系统的危害。当坏扇区出现在一个冗余卷上，卷管理工具程序尽可能地恢复并替换扇区。如果它不能替换扇区，就向NTFS返回一个警告，由NTFS来替换包含坏扇区的簇。假如磁盘不是以冗余卷的形式组织的，坏扇区上的数据就无法恢复了。从坏簇中可能读取到无法确定的数据。如果有些文件系统的控制结构驻留在坏扇区中，那么其中文件的某些数据，或是文件和文件组，甚至整个磁盘都有可能丢失。由于FAT文件系统并不对坏簇进行标记，因而极可能把它重新分配给另一个文件，从而引起新的问题。

虽然NTFS需要在卷管理工具的协助下才能进行坏簇数据的恢复，但是它最大程度地减少了扇区错误对文件系统所造成的损害。例如，当NTFS检测到坏扇区时，它能够重新映射扇区所在的簇，尽管簇上的数据丢失了，但是文件系统的其他部分仍然保持完整。同时，NTFS还对坏簇进行标记，防止对它的继续使用，从而最大限度地保护了用户的数据。

图5-42是NTFS坏簇重映射的示意图，包含坏扇区的簇1357将由一个新的簇1049替换。

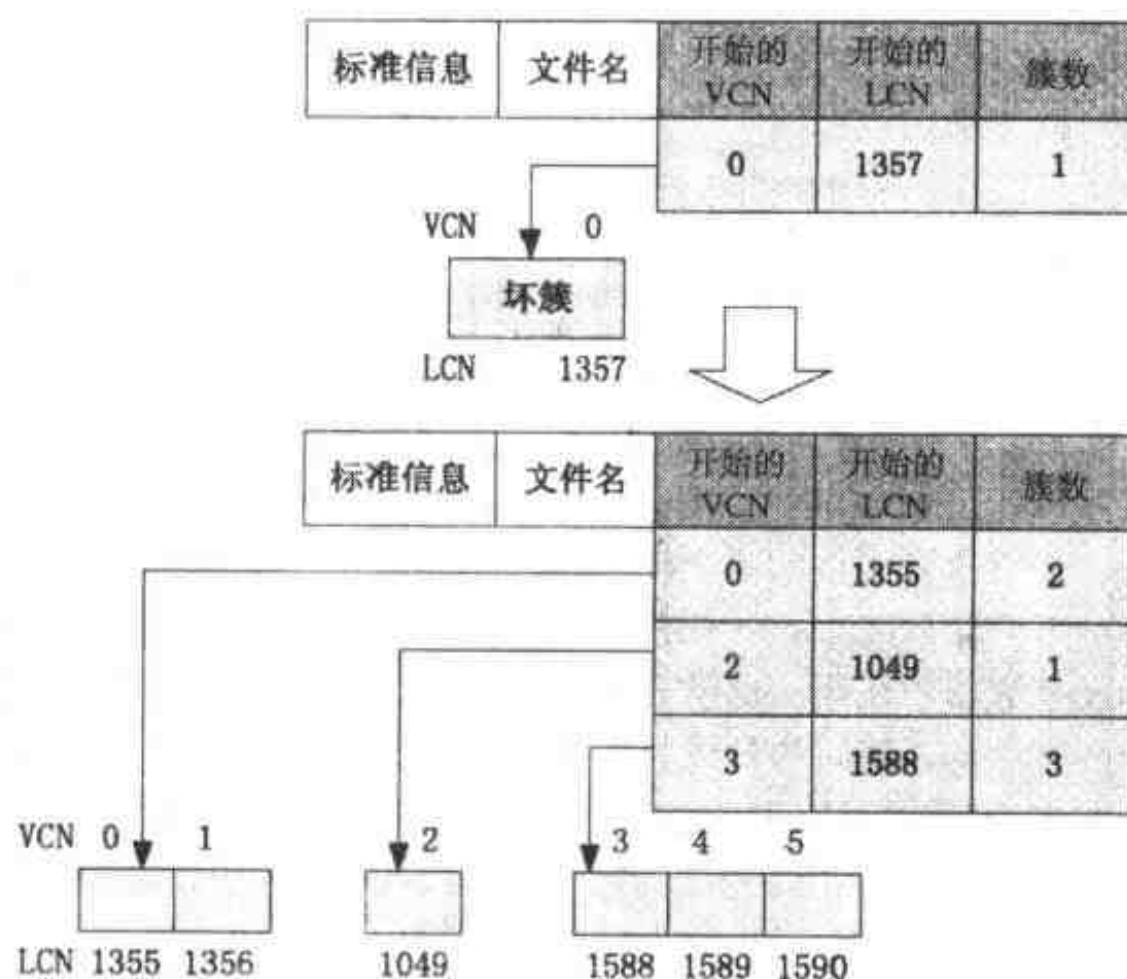


图5-41 坏簇重映射

如果存放文件系统数据的扇区发生错误，也依照同样的恢复过程。但是，在极少数情况下，文件损坏甚至可能发生在整个容错磁盘上。一个双重错误可能同时毁坏文件系统数据和重新构造它的方法。例如，如果在写入MFT副本时系统崩溃，镜像副本可能不会被完全更新，启动后重新引导又在主磁盘上发现坏扇区，并且与没有完全写到磁盘镜像上的位置相同，NTFS就不能从磁盘镜像上恢复数据。这时，NTFS执行一个特殊方案：如果它找到了不一致的数据，它将在卷文件中设置损坏位。当系统重新获得引导时，再调用Chkdsk程序来重新构造NTFS元数据。事实上，这种情况鲜有发生，因而也很少需要运行Chkdsk程序。但是NTFS考虑周全的设计使我们完全可以相信其极高的可靠性。

5.11 NTFS安全性支持

加密文件系统（Encrypted File System, EFS）提供的文件加密技术可将加密的NTFS文件存储到磁盘上。EFS特别考虑了其他操作系统上的现有工具引起的安全性问题，这些工具允许用户不经过权限检查就可以从NTFS卷访问文件。通过EFS，NTFS文件中的数据可在磁盘上进行了加密。EFS加密技术是基于公共密钥的，它用一个随机产生的文件密钥（File Encryption Key, FEK）通过加强型的数据加密标准（Data Encryption Standard, DES）算法——DESX对文件进行加密。EFS加密技术作为一个集成系统服务运行，易于管理，不易受攻击，并且对用户是透明的。如果用户要访问一个加密的NTFS文件，并且有这个文件的私钥，那么用户能够打开这个文件，并透明地将该文件作为普通文档使用。没有该文件私钥的用户对文件的访问将被拒绝。

DESX使用同一个密钥来加密和解密数据，这是一种对称加密算法（symmetric encryption）。

algorithm)。一般来说，这种算法的速度相当快，适用于加密类似文件的大块数据，但缺点也是很明显的：如果有人窃取了密钥，那么一切安全措施都形同虚设。而这种情况是很可能发生的：如果多个用户共享一个仅由DESX保护的的文件，每个用户都要求文件的FEK。如果不加密FEK显然是个严重的安全隐患；但是加密了FEK则要给每个用户同样的FEK解密密钥，这也是个严重的安全问题。

EFS使用基于RSA（Rivest Shamir Adleman）的公共密钥加密算法对FEK进行加密，并把它和文件存储在一起，形成了文件的一个特殊的EFS属性字段：数据解密字段（Data Decryption Field, DDF）。在解密时，用户用自己的私钥解密存储在文件DDF中的FEK，然后再用解密后得到的FEK对文件数据进行解密，最后得到文件的原文（即未加密的文件，与密文相对）。只有文件的拥有者和管理员掌握解密的私钥（private key）。任何人都可以得到加密的公共密钥，但是即使他们能够登录到系统中，由于没有解密的私钥，也没有办法破解它。Windows 2000/XP的最初版本把私钥存储在不是十分安全的磁盘上，随后的升级版本则将允许用户存储在某些便携设备上，例如智能卡上。尽管基于公共密钥的算法速度通常比较缓慢，但是EFS仅仅使用它来加密FEK，通过和加密文件的DESX配合，在使EFS的取得高速度的同时，也获得了令人羡慕的高安全性。

图5-42是EFS体系结构示意图。

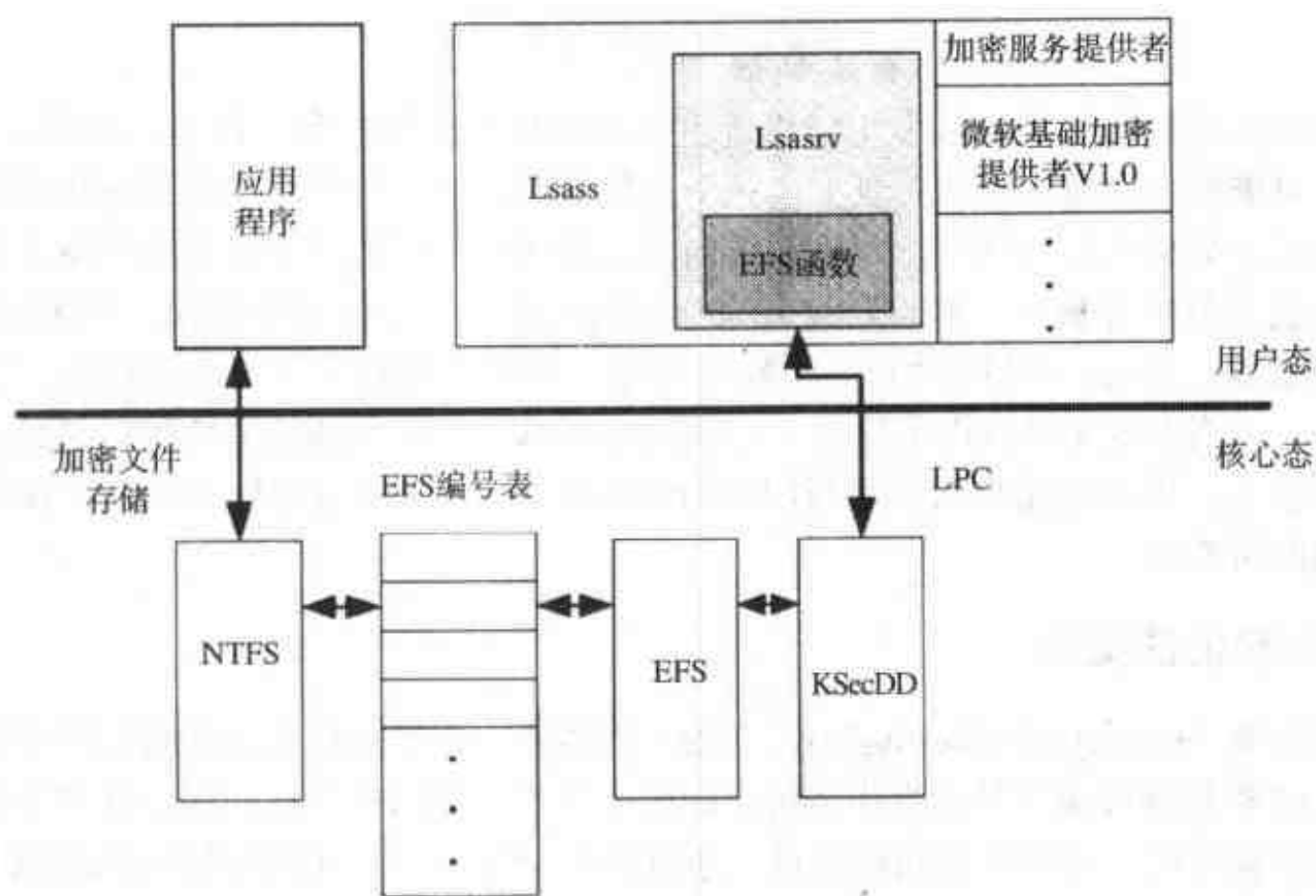


图5-42 EFS系统结构图

从图中我们可以看到，EFS的实现类似于在核心态运行的设备驱动程序，和NTFS有着十分紧密的联系。当处于用户模式的应用程序需要访问加密的文件时，它向NTFS发出访问请求，NTFS收到请求后立即执行EFS驱动程序。EFS通过KSecDD（\Winnt\System32\Drivers\KSecDD.sys）设备驱动程序转发LPC（local procedure call）给Lsass（Local Security Authority Subsystem——

\Winnt\System32\Lsass.exe)。Lsass不仅处理用户登录事务，也对EFS密钥进行管理。Lsass的功能组成部分Lsasrv (Local Security Authority Server—\Winnt\System32\Lsasrv.dll) 则侦听该请求，并执行所包含的相应的功能函数，在处于用户模式的加密服务API (CryptoAPI) 的帮助下，进行文件的加密和解密。

Lsasrv通过CryptoAPI来对FEK加密。通过动态链接库实现的CSP (cryptographic service provider) 极好地封装了加密服务API，以致于Lsasrv根本不必知道EFS算法的实现细节。Lsasrv取得EFS的FEK后，通过LPC返回给EFS驱动程序，然后EFS可以利用FEK通过DESX进行文件的解密运算，并通过NTFS把结果返回给用户程序。

下面我们深入探讨一下NTFS和EFS的交互关系以及Lsass是如何通过CryptoAPI对FEK进行管理的。

5.11.1 注册回调函数

尽管NTFS并不要求一定要挂接EFS驱动程序，但是离开EFS驱动程序，NTFS不能提供有关加密文件的操作。NTFS为EFS驱动程序(Winnt\System32\Drivers\Efs.sys)准备了一个插件接口，一旦EFS完成驱动初始化，就可以挂接到NTFS上。NTFS也为EFS提供了几个接口函数，通过它们，EFS可通知NTFS有关EFS存在及其相应API的信息。

5.11.2 首次加密文件

Windows 2000/XP通过在命令行程序cipher.exe或是目录的安全选项卡来加密文件。实际上，它们都是通过Advapi32.dll (Advanced Win32 APIs DLL)中的EncryptFile Win32 API进行的。Advapi32.dll调用Feclient.dll (File Encryption Client DLL)以取得Lsasrv中与EFS的交互接口。

当Lsasrv收到Feclient发来的加密文件LPC信息，它就使用Windows 2000/XP的模仿功能来模仿用户程序以便压缩文件，使得在Lsasrv上运行的文件压缩过程看起来就像是用户程序自己运行的。Lsasrv通常以系统账号运行。实际上，如果Lsasrv不模仿用户，就无权对相应文件进行压缩。

随后Lsasrv在文件所在卷的系统卷信息目录下创建记录压缩进程的日志文件记录。该文件名通常为Efs0.log。但是如有其他文件也正在压缩，则文件名中的0会依次递增，以保证该名的唯一性。

由于CryptoAPI需要用到用户注册配置信息，因此Lsasrv接着就通过Userenv.dll中的LoadUserProfile来载入所模仿用户的注册配置信息。最后，Lsasrv通过微软基础加密系统 (Microsoft Base Cryptographic Provider 1.0) 为文件产生一个基于RSA的FEK。

1. 构建密钥

在生成FEK以后，Lsasrv需要构建EFS信息并且把它作为一个文件属性和加密文件存储在一起。Lsasrv通过读取注册表中用户压缩文件产生的HKEY_CURRENT_USER\Software\Microsoft\WindowsNT\CurrentVersion\EFS\CurrentKeys\CertificateHash值来取得用户的公共密钥签名。Lsasrv通过该签名来取得用户的公共密钥和FEK。

接着，Lsasrv开始构建加密文件中的EFS 信息，作为加密文件的新属性。EFS把信息存储在

加密文件中一个包含多个密钥字段（key entry）的块(block)中，这些字段叫做数据解密字段（Data Decryption Field，DDF）。由于EFS允许多个用户共享加密文件，因而把密钥字段的集合叫做密钥链（key ring）。

图5-43清楚地显示了加密文件的EFS信息和密钥字段格式。EFS在密钥字段的第一部分存放了足够的信息来精确地标识用户的公共密钥，包括用户安全标识（security ID，SID）、存放密钥的容器名（container name），等等。关键字段的第二部分则包含有FEK的版本信息。

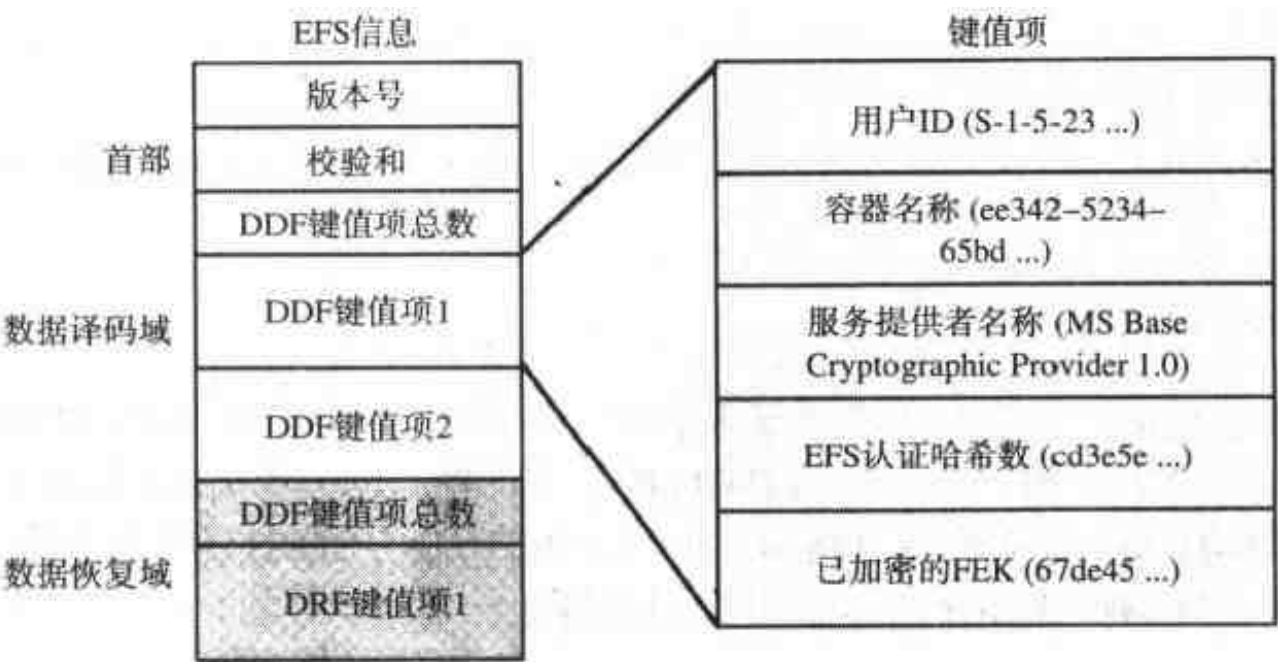


图5-43 EFS信息与键值项格式

随后，Lsasrv创建另一个密钥链以用来包含恢复密钥字段（Data Recovery Field，DRF）。DRF的格式和DDF相同。当用户丢失了其私钥或者有关管理部分需要访问用户数据时，DRF可以用指定帐号（designated account）和恢复代理（Recovery Agent）来解密文件。

最后，Lsasrv通过微软基础加密系统的MD5哈希功能来计算DDF和DRF的校验和，并把它存储在EFS信息头中。EFS通过检查这个总数来判断该加密文件的EFS信息的完好性。

2. 加密文件数据

在Lsasrv完成构建必要的压缩文件信息以后，就可以进行文件数据的压缩工作。

首先，Lsasrv建立一个备份文件Efs0.tmp（或是Efs1.tmp，依此类推），以防止压缩过程出现意外。Lsasrv对该文件应用严格的安全保护措施以保证除了系统管理员任何人都无权访问。然后Lsasrv初始化在第一阶段创建的日志文件。最后Lsasrv在日志文件中记录备份文件创建操作。在文件完全备份后，Lsasrv开始文件数据的压缩。

整个压缩过程，参见图5-44。

接着，Lsasrv通过NTFS向EFS驱动程序发送一个命令，以便将所创建的EFS信息增加到原文件中。NTFS收到命令，并不能自己处理，而需要调用EFS驱动程序。EFS驱动程序接受到来自Lsasrv的发送EFS信息，通过使用NTFS函数应用该信息。通过NTFS函数可以让EFS为NTFS文件增加\$LOGGED_UTILITY_STREAM属性。这时，执行回到Lsasrv。然后，Lsasrv将要加密的文件做一个完整备份，并且在日志文件中记录备份文件已是最新的。在这之后，Lsasrv向NTFS发送一个加密文件原来内容的命令。

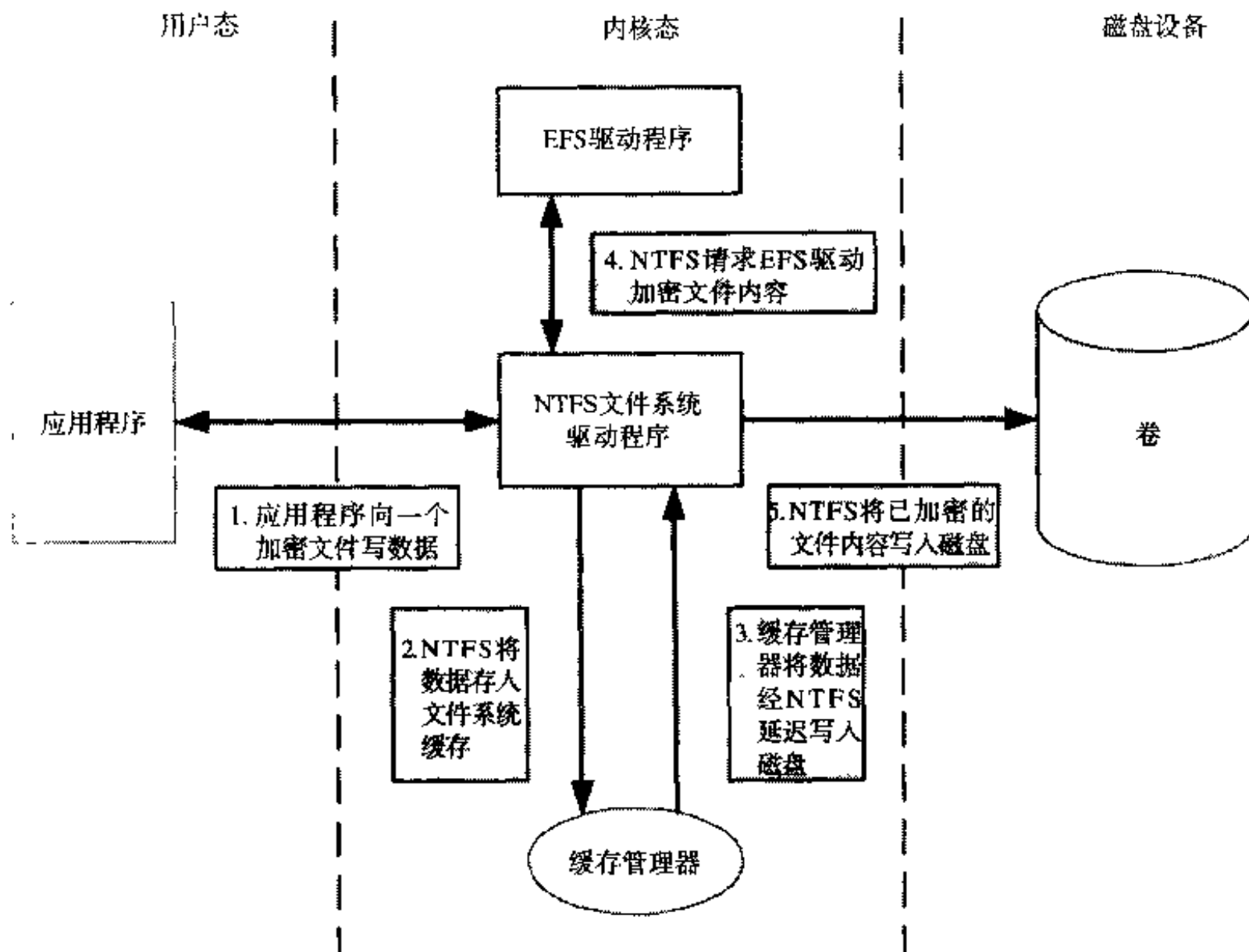


图5-44 EFS工作流程图

当NTFS收到EFS命令以加密文件时，NTFS删除原来文件的内容，并将备份数据拷贝到加密文件中。NTFS每拷贝一部分，NTFS就更新高速缓存中该文件的相应部分的内容。这也导致高速缓存管理器会通知NTFS以将文件数据写入到磁盘上。因为文件被标记为加密的，所以NTFS在写数据时会调用EFS来加密数据。EFS用NTFS传递的未加密的FEK来对文件进行DESX加密，加密是按扇区为单位进行的。

EFS完成文件数据的加密后，就在日志文件中记录下来，并删除备份文件。最后Lsassrv删除日志文件并返回控制权给要求进行文件加密的应用程序。

5.11.3 解密文件

一旦用户试图打开加密文件，就开始文件解密了。在打开文件时，NTFS先检查文件属性，再执行EFS驱动程序中的回调函数。EFS驱动程序读取与加密文件相关的\$LOGGED_UTILITY_STREAM属性，确保打开文件的用户具有访问权限（即DDF链或DRF链中的FEK对于该用户的私有/公共密钥），同时通过Lsassrv得到解密的FEK。

EFS本身并不能解密FEK，而需要靠Lsassrv来解密FEK。EFS通过Ksecdd.sys驱动程序向Lsassrv发送一个LPC信息以对\$LOGGED_UTILITY_STREAM属性中的FEK进行解密。当Lsassrv收

到LPC信息时，Lsasrv通过Userenv.dll（User Environment DLL）中的LoadUserProfile函数来将用户配置文件装入注册数据库中（如果以前尚未装入）。对EFS数据中的每个密钥域，Lsasrv通过用户的私钥来试图进行解密。如果成功，则Lsasrv返回一个解密的FEK；否则，该用户无法打开该文件。

1. 高速缓存解密FEK

加密一个FEK时，CryptoAPI会调用2000多个注册数据库API和400多个文件系统访问。因此，EFS驱动程序常常借助NTFS，通过高速缓存来避免这一开销。

2. 解密文件数据

在打开加密文件后，应用程序就可以读写文件了。当NTFS从磁盘上读数据时，NTFS先调用EFS来解密文件数据，然后再放入文件系统的高速缓存中。当NTFS往磁盘上写数据时，这些数据都处于未加密状态，一直到高速缓存管理器需要用NTFS来更新数据。这时，系统将调用EFS驱动程序来加密数据并写入磁盘。

5.11.4 备份加密文件

对任何文件加密功能的设计而言，文件数据对于可访问该文件的应用程序外，总是加密的。这一点尤其影响了备份工具。为此，EFS提供了一些特殊函数来支持备份，如OpenEncryptedFileRaw，ReadEncryptedFileRaw，WriteEncryptedFileRaw，CloseEncryptedFileRaw。通过这些函数，备份工具在备份时并不再需要对文件进行解密和加密。

习题

- 5.1 文件一般按什么分类？可以分成几类？
- 5.2 文件的物理结构有那几种类型？它们的优缺点各是什么？
- 5.3 文件目录的作用是什么？文件目录的表项应包括哪些内容？
- 5.4 文件目录有几种实现方式？它们的优缺点各是什么？
- 5.5 对文件和文件目录有几种操作方法？
- 5.6 分析和比较文件系统的三种模型：层次模型、VFS模型和Windows模型。
- 5.7 分析和比较文件系统的三种写入方式：谨慎写、延迟写和事务日志。
- 5.8 简述一下当今流行的文件系统的特点。
- 5.9 NTFS是如何维护文件系统的一致的？
- 5.10 NTFS是如何替换坏簇的？
- 5.11 NTFS是如何实现数据压缩的？
- 5.12 NTFS是如何实现文件数据的加密的？

第 ⑥ 章

I/O系统

第 ⑥ 章

I/O系统

6.1 I/O系统概述

操作系统中负责管理输入输出设备（即控制所有这些设备以完成执行期望的数据传送）的部分称为I/O系统，完成设备管理功能。

本章首先讲述I/O系统工作的重要性，然后分述I/O系统相关软件以及系统管理设备的主要策略，最后详细阐述Windows 2000/XP I/O系统设计和实现的特点。

6.1.1 设备管理的重要性

在计算机系统中，有大量的输入输出设备，其种类繁多，而且新设备也随着技术的发展而不断地出现。

那么这些输入输出设备在计算机系统中起什么作用呢？如果说，处理器和存储器是计算机系统的大脑的话，那么，输入输出设备就是计算机系统的五官和四肢。各种需要处理的信息和操作人员对计算机系统的操作命令，都要通过输入设备输入进计算机系统，处理后的信息和结果又要通过输出设备从计算机系统输出。输入输出设备是人机对话的界面和接口。

需要处理的信息和处理后的信息，它们的原始形态可能是各种物理信号量，如可见光、无线电波、红外线，也可能是各种有形介质如文稿、设计图、绘画、录音带、录像带等。输入输出设备对各种信息的载体不同，信号量的形态也不同，这才有可能造就计算机应用的多样性和普及性。可以说没有输入输出设备，就没有计算机的应用。

设备管理的重要性还表现在以下方面：

- 输入输出设备的性能经常成为系统性能的瓶颈。CPU性能越高，输入输出设备性能同CPU性能不匹配的反差也越大。如何解决这一矛盾，而又尽量不降低处理器的性能，是设备管理的一项重要任务。
- 输入输出设备千差万别，怎样对它们实现统一的管理，是设备管理的又一项重要任务。
- 在应用中，输入输出设备能否及时将各种信息传送给计算机系统，计算机发出的各种命令能否通过输入输出设备及时传送始执行部件，这些对于实时处理和控制系统而言，是至关重要的。

总之，输入输出设备是操作系统管理的重要资源之一，其重要性是不容低估的。

6.1.2 设备的分类

计算机设备种类繁多，因此可以从多个角度对设备进行分类。

1. 按设备的使用特性分类

按设备的使用特性分类，可分为存储设备、I/O设备、终端设备以及脱机设备等。

在I/O设备中，输入设备是计算机用以“感受”或“接触”外部世界的设备。这些设备可以是如温度计或雷达设备那样的装置，但更为常用的是穿孔卡片（纸带）读入机（或从电传打字机终端上读信息的设备）、键盘、鼠标等。换行输出设备是计算机用以“影响”或“控制”外部世界、或产生可读信息的设备，一般的有显示器、控制在电传打字机终端上打字的设备、硬拷贝输出类（如打印机、绘图仪、卡带穿孔机）等。在过去几十年中，卡片或纸带穿孔机是输入输出设备的典型代表。近年来，交互计算的发展使得这些卡带机（非交互设备）几乎完全从市场消失，其最普遍的取代物是各种CRT终端（交互设备）。

存储设备是计算机用来保存信息（通常称为写）的部件，写入以后还可以把这种信息取出来（读），如磁带、磁盘等各种外存设备。可移动的磁带或磁盘可以看作外存，在用于不同机器间传送数据时也可看作输入输出设备。

2. 按设备的信息组织方式分类

按信息组织方式来划分设备，可以把输入输出设备划分为字符设备（character device）和块设备（block device）。键盘、终端、打印机等以字符为单位组织和处理信息的设备被称为字符设备；而磁盘、磁带等以字符块为单位组织和处理信息的设备被称为块设备。

划分字符设备和块设备主要是依据设备的信息记录的大小，它的大小决定了设备一次操作的数据传送单位和内部是否可寻址。

块设备的基本特性是能够随时读写其中的任何一块而与所有其他块无关。

外存类设备都是块设备，因其记录长度通常为了一块，如磁盘、磁鼓。

字符设备传递或接受一连串的字符，不考虑任何块结构。它不可寻址，并且没有任何查找操作。终端、行式打印机、纸带、穿孔卡片、网络接口、小鼠标（供指示用）和大鼠标（供心理学实验用），以及除磁盘以外的大多数设备，都可看作字符设备。

通常，输入输出类设备都是字符设备，而存储类设备都是块设备。不过这种字符与块的分类方法是不严格而且不完整的，有些设备就不能按照这种分类方法分类。例如，时钟既不可以按块方式访问，也不产生或接受字符串，它所做的只是按规定好的时间间隔引起中断。

3. 按设备使用可共享性分类

按设备使用可共享性分类，可分为独占设备、共享设备和虚拟设备等。

独占设备是指在一个程序（作业、用户）的整个运行期间都必须由单个程序（作业、用户）独占，直至该程序（作业、用户）完成的设备，即在任一给定的时刻只能让一个进程使用。也就是说，必须保证一个进程对一个具体设备在可能相当长的时间内拥有唯一访问权。

例如打印机、卡片输入机、磁带驱动器等，都是独占设备，共享它们就很困难：如果几个用户同时使用一台打印机，把几个用户的打印任务随机地交织在一台打印机上是不行的。打印机必

须要在完成对一个进程的输出任务的完整处理后才能为另一进程服务。

独占设备的引入带来了一些问题：如果作业不是充分、连续地利用设备，独占分配就可能是低效率的，而且容易造成死锁。

共享设备是指能够同时让许多程序（作业、用户）使用的设备。例如磁盘就属于共享设备，多用户同时在同一磁盘上拥有打开的文件不会带来任何不良后果。不同进程向同一磁盘提出的读写操作一般能随意交叉。注意，共享设备有两种含义，广义的共享设备是指包括非并发和并发的共享，几乎所有的设备都是广义的共享设备。独占设备解释为只能顺序共享（即非并发共享）。狭义的共享设备是指并发共享，即操作系统中的共享设备的真正定义。

虚拟设备是指设备本身是独占设备，但经过某种技术处理，可以把它改造成共享设备，同时分配给多个进程。例如6.1.5节中讲解的SPOOLing技术。

4. 其他分类方法

按输入输出对象分：分为人机通信与机机通信设备。

按是否交互分：机机通信设备、外存、卡带机等属于非交互设备，终端为交互设备。

按数据传输速率分：分为高速设备和低速设备。上述人机通信类设备大都是低速设备（除了图形显示器等以外），而机机通信类设备大都是高速设备。这是由于人机通信速度受到人的操作速度限制。低速设备可以慢到每秒钟不到一个字符，比如人在键盘上的操作输入；高速设备可以快到每秒钟上百兆字节的传送，比如处理器芯片和高速硬盘之间的数据传输。

6.1.3 I/O设备的性能标准

I/O设备于系统的性能标准包括：

- 适应性和多样性（diversity）
- 容量（capacity）
- 响应时间（延迟，响应时间）
- 吞吐量（I/O吞吐量，I/O带宽）
- 代价（I/O代价 + CPU 开销）

较高的性能必须由硬件与软件配合才能实现。或者说，单单靠硬件的寄存器过程和连接过程，效率上还有很多有待改善之处，且安全保护得不到保证。

总之，设备硬件过程和各種硬件连接模式的复杂多样性导致了整个硬件I/O过程的复杂多样，且需要软件（操作系统）的进一步配合来达到更高性能。以后我们将看到，再加上用户要求的多样性，CPU进程划分的多样性，就最终导致了各种具体操作系统的I/O接口和数据结构与算法的多样性。

6.1.4 I/O系统的功能

I/O设备的使用者有用户程序（程序员和命令用户）和操作系统本身（即操作系统的I/O管理以外的其余部分）。设备使用（管理）的用户观点，即用户希望（实际）看到的设备使用（管理）接口和功能，也就相应地分为程序员观点和命令用户观点。

总的来说,用户对I/O设备的使用要求是方便、高效、安全和正确。相应地,设备管理功能也就应该在计算机硬件结构提供的既定设备范围及其连接模式下,完成用户对I/O设备的使用过程提供方便、提高效率和提供保护这三方面任务。下面我们分别从这三方面来详细分析用户的具体要求和相应设备管理功能的具体内容。

1. 方便性

用户总是希望方便地使用I/O设备,但遗憾的是,硬件I/O过程非常复杂,涉及到大量I/O细节,如寄存器、中断、控制字符、设备字符集(汉化、多国语言国际化)等。如果让用户直接使用设备硬件接口,那么就会带来以下问题:

1) 不可能是方便的,相反,繁琐到程序员可能要把大部分精力放到这些与应用本身没有直接关系的I/O细节上,而不能集中专心于应用本身,工作速度受到极大影响。

2) 重复劳动:重复的知识了解、设计、编码和输入,这些都导致了工作效率低和系统代价大(因重复占用内存和外存)。

3) 变化(如程序使用要求的变化,系统配置的变化等)将会引起程序改变(至少需要重新编译),影响了程序的独立性和适应性。

4) 考虑到I/O设备类型的众多和多变以及I/O连接模式的复杂多样,还有两者的迅速发展导致的更复杂的设备和技术的不断出现,实际上,让用户或程序员来了解和处理程序中使用的每个设备的、数量庞大且不停更新的实际物理过程操作细节已成为不可能的事。

5) 让程序员直接操纵设备,不利于设备及其上数据的保护。

6) 显然用户要想方便,要想避免上述诸多弊病,就不要关心硬件I/O过程中在系统内部运行的I/O细节。

为了弥补硬件接口给用户带来的不便,操作系统的设备管理部分应该做好以下工作:

1) 其设备管理的第一大功能即是提供简便易用的高级逻辑接口,并接受和翻译之。这些接口由对逻辑设备进行的逻辑操作组成。

2) 实现抽象接口到物理接口的转化,即将高级逻辑操作转化为低级物理操作,将逻辑设备和逻辑性质转化为物理设备和物理性质,以便掩蔽设备的硬件物理操作和组织的细节。

另外,抽象接口除掩蔽硬件细节外还要掩盖依赖于硬件的软件技术细节,这两种细节合称为机器相关细节,即依赖于机器的细节。设备抽象接口是由设备管理功能接口和文件系统功能接口共同提供的,更准确地说,设备抽象接口包含在文件系统统一接口中。抽象接口使广义的设备独立性成为可能,向用户展示一个大大简化了的计算环境观点,同时,抽象接口也是提高效率技术的前提。

具体来说,需要掩蔽的硬件细节有六个方面。

1) 与接口寄存器相关的I/O操作过程。这个过程由准备、发出命令、数据传送、控制状态(含轮询或中断)及错误处理等构成。

2) 错误处理是I/O软件的另一重要课题。总的来说,错误应当在尽可能接近硬件的层次处理。如果控制器发现一个读数错,它就该尽其所能,努力自行纠正它。倘若控制器不能处理该错误,就应当交给设备驱动程序进行处理,也许只要再读这一块即可。很多错误是偶然的,例如读头的

尘埃导致读错，重复该操作，错误就会消失。仅当底层不能正确处理错误时，才上交给高层处理。

3) ESC码与各种字符集之间编码的变换。

4) 掩蔽设备的物理细节和不同设备的物理性质的差别，提供抽象的统一的设备逻辑性质，如逻辑块长与物理块长。我们把为了更有效地传送数据而将几个逻辑记录合为一个物理记录的做法称为组块 (blocking)。再如自由块的分配与释放和相应的存储分配算法，掩蔽软件存取机制。块设备的物理性质是由文件系统的逻辑文件、逻辑记录和目录等来掩蔽的。

5) 具体设备、虚设备与文件系统的统一接口。

同一程序在不同次运行中，所使用的设备很可能是不同的，这是由于不同次运行的不同需要及系统配置的改变。这时可能有三种情况，即改程序、重新编译或重定向。广义的I/O设备独立性是指整个设备管理功能的抽象接口，程序不关心物理设备的操作和组织细节，而只看到具有逻辑名称和逻辑性质的逻辑设备和逻辑操作，不关心具体I/O设备是哪一个具体设备或是哪一个文件，设备改变而程序不变，不必修改，也不必重新编译。设备改变所需的改变是很大的，有些设备确实不同，需要差别很大的设备驱动程序，但用户不必关心，这一事实所带来的问题应该由操作系统来管理。所使用的设备的变化对程序是透明的。

为了实现设备独立性，操作系统通常提供虚（逻辑）设备抽象接口。

I/O与文件系统接口统一的必然性体现在，用户程序经常需要在文件和设备间变换I/O方向；可能性体现在对文件的访问与对设备的访问在接口上有共性可循。

块设备的设备独立性是由文件系统的文件目录概念提供的。

输入输出设备可以抽象地看作一个顺序数据的源和目标 (sink)。在每个用户面前表现的，是一个可以被一组类似于为使用文件而提供的简单通用操作所控制的虚设备的集合。事实上，如果假设对文件的存取都是顺序的，虚设备与文件没有什么区别，那么虚设备可以用那些为存取文件而提供的相同（逻辑）指令来操纵。

6) 设备控制模式。总体上看，计算机系统对输入输出设备的控制模式有四种方式：程序查询（轮询）方式、程序中断方式、输入输出通道和直接传送方式（DMA）。

对I/O设备的程序轮询的方式，是早期的计算机系统对I/O设备的一种管理方式。处理器定时对各种设备轮流询问一遍有无处理要求，有要求则加以处理，完成后返回继续工作。尽管轮询需要时间，但它通常还是比I/O设备的速度要快得多，所以一般不会发生不能及时处理的问题。当然再快的处理器能处理的输入输出设备的数量也是有一定限度的。而且程序轮询，毕竟占据了相当一部分CPU处理时间，因此它是一种效率较低的方式。在现代计算机系统中已很少应用。

中断，即异步中的同步技术，是操作系统的I/O管理中的一个关键问题。它与相关的冻结和并发操作，都是从用户程序中看不到的、而在实际运行过程中存在的内部现象和技术，将它们隐藏起来的正是操作系统。中断是生活中不愉快的事情，应当把它们深埋在系统的中心，只让系统尽可能少的部分知道。

输入输出通道方式和DMA方式，是在I/O设备中大量采用的技术，将在后面具体分析讨论。

此外，虽然掩蔽了硬件细节，还要在接口中提供方便，以便能了解系统内部的实际情况，如资源使用情况、使用者情况等。

2. 效率

用户永远关心效率：非系统用户关心使用效率（即程序运行效率与操作效率）；系统用户关心系统利用率、系统代价、系统工作效率和用户效率信誉。对普通用户是程序不希望等待设备；对系统用户是设备不希望等待程序。在本章的讨论中，我们会看到大量的技术被引入以提高设备与CPU的效率，如中断、缓冲、DMA、通道等，这些技术不仅需要相应的硬件，还需要相应的软件来与硬件配合，才能达到引入这些技术所预期的目标。另外，在此基础上，还需要用纯软件的技术来进一步挖掘效率和利用率的潜力。所以，操作系统的设备管理功能的第二大任务是采用各种纯软件或与硬件配合的软件技术来提高设备效率和与此相关的系统效率，提供物理I/O设备的共享并优化这些设备的使用，同时借助抽象接口使得这些效率优化技术得以由系统在内部实施并对用户透明。

第二个任务（即效率技术）与抽象接口之间有紧密的联系：前者要借助后者，才能在从高级到低级的转换过程中实施效率技术。若无抽象接口，则无高级至低级的转换，而是直接操纵，效率优化技术就无机可入了。总之，没有抽象接口，即为用户自己采用效率技术；而反之，由系统采用效率技术，即为抽象接口。抽象接口是效率技术的前提。

针对第二个任务，操作系统设备管理功能要实现和采用以下技术。

(1) 并发

并发导致了一系列问题：命名与保护、登记（UCB）、实际分配或映射、回收、挂起唤醒与调度；并发还使得进一步提高设备利用率等的效率技术得以采用，如共享设备技术、虚（逻辑）设备。

设备分配技术与策略：可共享设备只适用于前两种分配技术，即共享设备技术和独占设备技术，但考虑性能要尽量用前者，即共享设备技术来分配；独占设备只适用于后两种分配技术，即独占设备技术和虚拟设备技术，且尽可能采用后者，即虚拟设备技术。

1) 共享设备技术。数据和程序的并发共享带来效率和方便。为了在解决设备与CPU的速度匹配问题上使设备和CPU的利用率达到最大程度，即优化性能，系统希望根据每个设备的特征来全局调度安排设备的操作，设备的并发共享技术使这种全局调度安排成为可能。例如，对一个可移动磁盘的读写要求它们可以排序，以使操作间读写头的移动距离尽可能短。

注意，在采用共享设备技术时，要考虑设备类型，即共享设备技术只适于可共享设备，不适用于独占设备。

2) 独占设备技术。

3) 虚拟设备技术。虚拟设备技术的作用是提高慢速独占设备的利用率，采用了假脱机技术后一个程序对慢速独占设备的占用时间短了（因为连续了）。

(2) 级冲

此处的缓冲与前述的缓冲虽然都是为了不等待，但有三点不同：

- 1) 前述为用户程序内并发，此处为用户程序内的串行。
- 2) 前述为用户程序中采用的技术，此处为操作系统采用的技术。
- 3) 虽然都是为了同步和不等待，但前述采用测试技术，此处采用中断。

缓冲是一个广泛的概念和广泛采用的技术。就广义来说，缓冲就是在通信问题中为了通信双

方的速度匹配而引入的一个中间层次，这个层次的速度比通信双方中较慢的一方快，而与较快的一方更匹配。但是这个层次不能完全代替较慢的一方，因为它有各种局限，例如有时是成本原因，有时是信息表现形式不同，（如磁盘不能代替纸）等。

按照这个定义，缓冲的内涵变化非常大，就其设置的位置可分四种：

1) CACHE。

2) I/O设备或控制器内部的纯硬件缓冲区，如打印机内部的硬件缓冲区（可在打印机手册上看到有关说明），再如磁盘控制器上的纯硬件缓冲区。

3) I/O在内存开设的缓冲区，如操作系统在内存开设的I/O缓冲区和文件系统缓冲区，如早期的用户程序自设的单双多缓冲区等。

4) 脱机与假脱机技术实质上属缓冲技术，实际上是为慢速I/O设备在外存开设的缓冲区。

前两种由于是纯硬件管理的，即缓冲区的读写和设置完全由硬件来做，故称为纯硬件的缓冲区。后两种由于需要软件的管理，即缓冲区的读写由程序指令来做，故称为软件缓冲区。

保证采用缓冲技术的有效性有两个前提：

1) 有阵发性（包括突发性与间歇性）的快，持续可接受的慢。所有的I/O缓冲区正是在这个前提下采用的，而CACHE则不是。

2) 访问局部性，这适用于可多次访问的存储介质，CACHE正是在这个前提下采用的，而I/O缓冲区则不是。注意，对于这种可重用的I/O数据（即存储介质），存在数据一致性（即缓冲区与磁盘上内容的一致性）问题。

缓冲的目的虽然都是为了速度匹配，但也可细分为很多种：

1) 减缓单纯的等待，如无中断前的单双多缓冲区。

2) 在通过引入中断技术已完全消除了相互等待的前提下，进一步减少中断次数和降低程序响应时间，如无DMA前的盘控制器硬件缓冲区、打印机硬件缓冲区等。

3) 在通过引入中断技术已完全消除了相互等待的前提下，虽然不能进一步减少中断次数，但可以提高设备速度（效率），如采用了DMA后的盘控制器硬件缓冲区。

4) 减少程序执行时间，提高异步程度，如操作系统缓冲区。

操作系统管理中也有多处采用了缓冲技术，如CACHE和文件缓冲池，以及此处的I/O缓冲技术等。

3. 保护

用户希望能安全正确地使用设备，这种希望体现为由设备传送或管理的数据应该是安全的，不被破坏和泄密的；对设备拥有所有权的用户，这种希望体现为设备不能被破坏，系统在安全可靠方面的信誉应得到保证。操作系统的设备管理功能的另一任务是提供安全保护机制，用于保护设备，尤其是保护由设备传送或管理的数据。

6.1.5 设备分配

在计算机系统中，设备、控制器和通道等资源是有限的，并不是每个进程随时都可以得到这些资源。进程首先需要向设备管理程序提出申请，然后由设备管理程序按照一定的分配算法给进

程分配必要的资源。如果进程的申请没有成功，就要在资源的等待队列中排队等待，直到获得所需的资源。

下面就来讨论与设备分配相关的数据结构，以及设备分配的一些原则与策略。

1. 设备控制表与设备等待队列

为了记录系统内所有设备的情况，以便对它们进行有效的管理，引入了一些表结构，如为每个设备（通道、控制器）配置的设备（通道、控制器）控制表等。由于系统的管理、分配方式不同，实际采用的表结构也不相同，例如通道控制表就只有在采用通道控制方式的系统中才会出现。

同时，系统保留一张系统设备表，每个表项对应一个物理设备，用来记录所有已经连接到系统中的设备的情况。与设备分配有关的数据结构如图6-1所示。

设备（通道、控制器）等待队列是由等待分配资源的进程控制块组成的，其组织方式可以按照先来先服务（FCFS）的顺序，也可以按照优先级顺序。

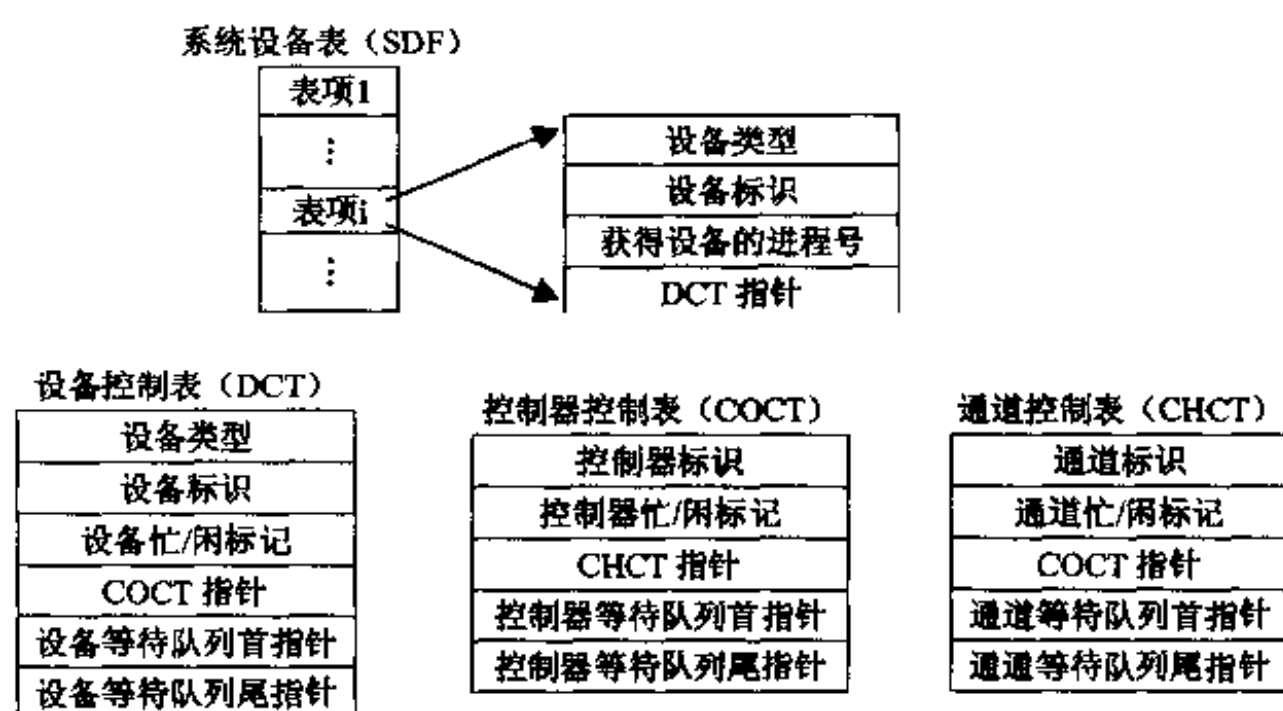


图6-1 设备管理的数据结构

2. 设备分配的原则

设备分配的总原则是，一方面要充分发挥设备的使用效率，同时又要避免不合理的分配方式造成死锁、系统工作紊乱等现象，使用户在逻辑层面上能够合理方便地使用设备。

- 考虑设备的特性和安全性。设备的特性是设备本身固有的属性，一般分为独占、共享和虚拟设备等，这在前面对设备的分类中已经做了说明。对不同属性设备的分配方式是不同的，本节稍后分别进行一些简单介绍。

从安全性方面考虑，有安全分配方式和不安全分配方式两种。在安全分配方式中，每当进程发出I/O请求后就进入阻塞状态，直到其I/O操作全部完成时才被唤醒。进程需要的所有资源必须一次性进行分配，分配后即归进程所有，直到操作结束。这种方式排除了死锁“请求和保持”的必要条件，因而是安全的，但是效率比较低，CPU与I/O设备串行工作。而在不安全分配方式中，进程发出I/O请求后继续运行，如果需要还可以发出其他的I/O请求，申请到的设备一旦使用完就立即释放，仅当请求的设备已经被其他进程占用的时候才进入阻塞状态。这种方式提高了运行效

率，但是存在造成死锁的可能，因此设备分配程序中应该增加预测死锁的安全性计算，这在一定程度上增加了程序的复杂性。

- 设备分配策略。与进程的调度相似，设备的分配也需要一定的策略，通常采用先来先服务（FCFS）和高优先级优先等。

先来先服务，就是当多个进程同时对一个设备提出I/O请求时，系统按照进程提出请求的先后次序，把它们排成一个设备请求队列，并且总是把设备首先分配给排在队首的进程使用。

高优先级优先，就是给每个进程提出的I/O请求分配一个优先级，在设备请求队列中把优先级高的排在前面。如果优先级相同，则按照FCFS的顺序排列。这里的优先级与进程调度中的优先级往往是一致的，这样有助于高优先级的进程优先执行、优先完成。

3. 独占设备的分配与虚拟设备

独占设备每次只能分配给一个进程使用，这种使用特性隐含着死锁的必要条件，所以在考虑独占设备的分配时，一定要结合有关防止和避免死锁的安全算法。

系统中的独占设备是有限的，往往不能满足诸多进程的要求，会引起大量进程由于等待某些独占设备而阻塞，成为系统中的“瓶颈”。另一方面，申请到独占设备的进程在其整个运行期间虽然占有设备，利用率却常常很低，设备还是经常处于空闲状态。为了解决这种矛盾，最常用的方法就是用共享设备来模拟独占设备的操作，从而提高系统效率和设备利用率。这种技术就称为虚拟设备技术，实现这一技术的软、硬件系统被称为假脱机（Simultaneous Peripheral Operation On Line, SPOOL）系统，又叫SPOOLing系统。

SPOOL系统通常分为输入SPOOL和输出SPOOL，两者工作原理类似。下面就以常见的共享打印机为例，说明输出SPOOL的基本原理。

打印机是一种典型的独占设备，引入SPOOL技术后，用户的打印请求传递给SPOOL系统，而不是真正把打印机分配给用户。SPOOL系统的输出进程在磁盘上申请一个空闲区，把需要打印的数据传送到里面，再把用户的打印请求挂到打印队列上。如果打印机空闲，就会从打印队列中取出一个请求，再从磁盘上的指定区域取出数据，执行打印操作。由于磁盘是共享的，SPOOL系统可以随时响应打印请求并把数据缓存起来，这样就把独占设备改造成了共享设备，从而提高了设备的利用率和系统效率。

4. 共享设备的分配和磁盘调度策略

磁盘是典型的共享设备。在用户处理的信息量越来越大的情况下，对磁盘等共享设备的访问也越来越频繁，因而访问调度是否得当直接影响到系统的效率。

首先，必须了解磁盘的结构。盘面上有一系列同心圆，用来记录信息，称为磁道。同时，磁盘盘面一般都被划分成若干相等的扇形，将每条磁道分割成许多弧段，一个弧段就称为一个扇区。尽管扇区由于所在的磁道不同，其实际的长度也不相同，但是它们记录的信息量都是相等的。计算机系统硬盘，由若干双面记录信息的磁盘和读写磁盘的磁头组成，有两种基本类型：固定头磁盘和移动头磁盘。固定头磁盘盘面上的每一条磁道都有一个读写头，而移动头磁盘每个盘而只有一个读写头，执行磁盘操作的时候必须首先移动磁头找到正确的位置。固定头磁盘访问速度快，但成本较高；移动头磁盘访问速度相对较慢，但成本也比较低，是通常采用的类型。

对磁盘上的物理信息定位需要三个参数：柱面（磁道）、磁头（盘面）和扇区，其中柱面是由各个盘面上位置相同的磁道所组成的，通常由外向里从0开始依次编号。访问磁盘需要的时间由三部分组成：磁头移动时间、旋转延迟时间和传送时间。磁头移动时间是指磁头从当前位置移动到正确磁道上所需要的时间，又称为寻道时间；旋转延迟是指磁盘从当前位置旋转到需要的物理块的时间；传送时间是指实际对磁盘访问操作所需要的时间。由于前两部分都涉及到机械运动，所以磁盘访问速度与计算机系统内部处理速度比较起来就慢了很多。访问时间中磁头移动又占了相当大的比例，大约为70%。这样，如何调度磁头的移动就成为操作系统中提高磁盘访问速度的关键，而所谓的磁盘调度策略主要是指对磁头移动的调度。

磁盘调度策略主要有下面列举的一些。

- 先来先服务（FCFS）。这是最简单的一种调度策略，根据的就是进程对磁盘提出访问请求的先后次序。这种策略最大的优点是公平、简单，所有进程的请求都能够依次满足，不会出现“饥饿”的现象。

在用户请求均匀分布的情况下，FCFS策略就相当于随机访问，没有对访问进行任何优化。当访问请求比较多时，这种策略对于设备吞吐量和响应时间会产生不良影响，平均寻道距离也会比较大。因此这种策略仅适用于磁盘I/O请求比较少的情況。

- 最短寻道时间优先（Shortest Seek Time First, SSTF）。这种策略每次都选择要求访问的磁道与磁头当前所在的磁道距离最近的请求，优先予以响应。采用这种策略，可以保证每次寻道时间最短，对于提高设备吞吐量也有一定好处。其缺点是对用户来说，请求被响应的机会不均等，中间磁道的访问较为优先，而越偏离中心的磁道访问响应越差。在请求很多的情况下，对内、外边缘磁道的访问可能被无限期拖延，造成“饥饿”现象。
- 扫描（SCAN）。Denning首先提出了这种策略，其目的就是克服SSTF策略的缺点，在考虑访问磁道距离时更优先考虑磁头当前移动的方向。例如，当磁头正在从里向外移动时，SCAN策略响应的下一个请求应该是要访问的磁道在当前磁道以外且距离最近。就这样一直到再没有向外方向上的请求时，磁头移动的方向才会改变，变为从外向里。这时类似地，响应的下一个请求应该是要访问的磁道在当前磁道以里且距离最近，直到再没有向里方向上的请求。这样，SCAN策略就基本上克服了SSTF策略中的“饥饿”现象，而且具有SSTF策略的优点，但是两侧磁道的访问频率仍然会低于中间磁道。由于这种策略中磁头移动的规律与电梯运行的规律极为相像，所以也经常称为电梯策略。
- 循环扫描（Circular SCAN, CSCAN）与N步扫描。循环扫描是对扫描策略的一种改进，不同之处在于磁头不是往返扫描访问，而是只沿一个方向反复扫描。例如只从里向外，当磁头移动到最外面一个被访问的磁道以后，不反向扫描访问，而是直接移动到最里面一个需要访问的磁道上，仍旧从里向外扫描。

N步扫描策略则是在SCAN的基础上，将磁盘请求队列分成若干个长度为N的子队列，磁盘调度按照FCFS策略依次处理这些子队列，而在每个子队列内部则按照SCAN策略进行处理。当N值取得很大时，N步扫描策略的性能接近于SCAN策略；而当N=1时，N步扫描策略就退化为FCFS策略。

这两种策略具有SCAN策略的优点，并在一定程度上消除了其缺点。根据模拟研究的结果，在磁盘访问负荷较小的情况下，SCAN策略是最好的；而在中等以上负荷的情况下，循环扫描策略效果最佳。

6.1.6 I/O系统功能的实现

1. 结构与过程概述

关于I/O软件结构，基本思想在于把软件组织成为一系列的层，较低的层参与隔离硬件特征，而较高的层则参与向用户提供一个友好的、清晰而规整的接口。这种分层思想旨在以一种易于领会和有效的方式实现I/O管理的功能和目标。I/O软件一般共分四层：中断处理程序，设备驱动程序，与设备无关的操作系统软件，以及用户级软件（指用户空间的I/O软件）。

这四层与操作系统结构有相似之处。从功能上看，设备无关层是I/O管理的主要部分；从代码量上看，驱动层是I/O管理的主要部分。注意分层的相对灵活性，各层之间的界面并不是死的。出于效率或其他考虑，上层中的某些功能实际放在下层中完成，如中断时的驱动、驱动层中的某些设备无关处理等，故各层之间的确切界面是依赖于具体系统的。

2. 接口（设备管理功能的使用）

如前所述，设备管理功能的接口是设备硬件的一个大大简化了的简单抽象的接口，提供的是对具有逻辑性质的逻辑设备上的逻辑操作。由文件系统和设备管理功能接受、翻译、转换为相应的物理设备、物理性质、物理操作。这种接受、翻译和转换是设备管理功能的主要任务之一。

1) 设备管理功能与文件系统的接口统一，以实现完全彻底的设备独立性，即当I/O方向在设备与文件间变化时也不改变程序。这种设备独立性和接口一致性体现在多个方面。

2) 设备命名，一个文件或一台设备的名称只应该是一个字符串或一个整数，它不以任何方式依赖于设备。在UNIX中，软盘、硬盘和其他所有块设备都能安装在文件系统层次中的任意位置。因此，用户不必了解哪个名字相当于哪台设备。所有文件和设备都用相同的方法（即路径名）进行访问。如何给诸如文件和I/O设备这样的对象命名，是操作系统中的一个重要课题。

与命名机制密切相关的是保护。系统如何阻止用户访问他们无权访问的设备呢？微机系统多半根本不设保护，任何进程能够做它想要做的任何事情。在大多数主机系统中，用户进程对I/O设备的访问完全被禁止。在UNIX中，采用的是一种比较灵活的方案：对应于I/O设备的特别文件受一般的rwx位保护。系统管理员据此为每台设备确定适当的授权。

3) 接口中的逻辑记录与逻辑操作，返回信息。有些系统的接口中可能没有逻辑记录的概念，此时用户看不到“块”，更不用说块长了。

4) 系统调用接口可能反映为函数库（即被函数库所掩蔽）。

5) 独占设备的管理（分配与释放）与设备的打开关闭、设备的安装、开机连接（即动态配置）设备名的查找（硬连线），等等。

6) 设备接口与文件系统接口可能的不同有：打开设备的操作名不叫Open，而叫Attach或Allocate；关闭设备的操作名不叫Close，而叫Detach或Deallocate。

与文件打开关闭操作的理由相同，由于一个设备并不永远属于一个程序，为了系统管理方便

或存取方式，程序在使用设备时必须在首尾使用开关操作，以建立或断开程序与逻辑设备间的逻辑联系。建立联系即使用户保留指定设备，执行从符号到实际物理设备的映射；断开联系即收回设备。

开关操作的用法与文件相同，只是操作名不同而已。

6.2 I/O软件的组成

设备管理软件的设计水平决定了设备管理的效率。从事I/O设备管理软件的结构，其基本思想是分层构造，也就是说把设备管理软件组织成为一系列的层次。其中低层与硬件相关，它把硬件与较高层次的软件隔离开来。而最高层的软件则向应用提供一个友好的、清晰而统一的接口。

6.2.1 I/O软件的目标

1. 设备独立性

设计I/O软件的一个最关键的目標是设备独立性（device independence）。也就是说，除了直接与设备打交道的低层软件之外，其他部分的软件并不依赖于硬件。

I/O软件独立于设备，就可以提高设备管理软件的设计效率。当输入输出设备更新时，没有必要重新编写全部设备。在实际应用中我们看到，在一些操作系统中，只要安装了相对应的设备驱动程序，就可以很方便地安装好新的输入输出设备。甚至不必重新编译就能将设备管理程序移到他处执行。

I/O设备管理软件一般分为四层：分别是中断处理程序、设备驱动程序、与设备无关的系统软件 and 用户级软件。至于一些具体分层时细节上的处理，是依赖于系统的，没有严格的划分，只要有利于设备独立这一目标，可以为了提高效率而做出不同的结构安排。

2. 统一命名

操作系统要负责对输入输出设备进行管理。有关管理的一项重要工作就是如何给I/O设备命名。不同的系统有不同的命名原则。对设备统一命名，是与设备独立性密切相关的。这里所说的统一命名，是指在系统中采取预先设计的、统一的逻辑名称，对各类设备进行命名，并且应用在同设备有关的全部软件模块中。

通常的做法是，用一个序列字符串或一个整数来表征一个输入输出设备的名字。这个统一命名不依赖于设备，也就是说在同一个设备的名称之下，其对应的物理设备可能发生了变化，但它并不在该名称上体现，因此用户并不知晓。例如在UNIX中，软盘、硬盘和其他所有块设备都能安装在文件系统层次中的任意位置。因此，用户不必知道哪个名字对应于哪台设备。所有文件和设备都用路径名来检索。又如，一个软盘可以安装到目录/usr/ast/backup/Monday下，所以拷贝一个文件到/usr/ast/backup/Monday就是将文件拷贝到软盘上。这样，所有文件和设备都使用相同的方式进行定位。

6.2.2 中断处理程序

中断处理程序在设备管理软件中是一个相当重要的部分。本节着重分析中断处理程序的内在

工作方式，然后讨论中断在设备管理中的作用。

1. 中断的基本概念

中断是指计算机在执行期间，系统内发生任何非寻常的或非预期的急需处理事件，使得CPU暂时中断当前正在执行的程序而转去执行相应的事件处理程序，待处理完毕后又返回原来被中断处继续执行或调度新的进程执行的过程。引起中断发生的事件被称为中断源。中断源向CPU发出的请求中断处理信号称为中断请求，而CPU收到中断请求后转到相应的事件处理程序称为中断响应。

在有些情况下，尽管产生了中断源和发出了中断请求，但CPU内部的处理器状态字PSW的中断允许位已被清除，从而不允许CPU响应中断。这种情况称为禁止中断。CPU禁止中断后只有等到PSW的中断允许位被重新设置后才能接收中断。禁止中断也称为关中断，PSW的中断允许位的设置也被称为开中断。开中断和关中断是为了保证某段程序执行的原子性。

还有一个比较常用的概念是中断屏蔽。中断屏蔽是指在中断请求产生之后，系统有选择地封锁一部分中断而允许另一部分中断仍能得到响应。不过，有些中断请求是不能屏蔽甚至不能禁止的，也就是说，这些中断具有最高优先级，只要这些中断请求一旦提出，CPU必须立即响应。例如，电源掉电事件所引起的中断就是不可禁止和不可屏蔽的。

2. 中断的分类与优先级

根据系统对中断处理的需要，操作系统一般对中断进行分类并对不同的中断赋予不同的处理优先级，以便在不同的中断同时发生时，按轻重缓急进行处理。

根据中断源产生的条件，可把中断分为外中断和内中断。外中断是指来自处理器和内存外部的中断，包括I/O设备发出的I/O中断、外部信号中断（例如用户键入ESC键）。各种定时器引起的时钟中断以及调试程序中设置的断点等引起的调试中断等。外中断在狭义上一般被称为中断。

内中断主要指在处理器和内存内部产生的中断。内中断一般称为陷阱（trap）或异常。它包括程序运算引起的各种错误，如地址非法、校验错、页面失效、存取访问控制错、算术操作溢出、数据格式非法、除数为零、非法指令、用户程序执行特权指令、分时系统中的时间片中断以及从用户态到核心态的切换等都是陷阱的例子。

为了按中断源的轻重缓急处理响应中断，操作系统为不同的中断赋予不同的优先级。例如，在UNIX系统中，外中断和陷阱的优先级共分为8级。为了禁止中断或屏蔽中断，CPU的处理器状态字PSW中也设有相应的优先级。如果中断源的优先级高于PSW的优先级，则CPU响应该中断源的请求；反之，CPU屏蔽该中断源的中断请求。

各中断源的优先级在系统设计时给定，在系统运行时是固定的。而处理器的优先级则根据执行情况由系统程序动态设定。

除了在优先级的设置方面有区别之外，中断和陷阱还有如下主要区别：

- 陷阱通常由处理器正在执行的现行指令引起，而中断则是由与现行指令无关的中断源引起的。
- 陷阱处理程序提供的服务为当前进程所用，而中断处理程序提供的服务则不是为了当前进程的。

- CPU执行完一条指令之后，下一条指令开始之前响应中断，而在一条指令执行中也可以响应陷阱。例如执行指令非法时，尽管被执行的非法指令不能执行结束，但CPU仍可对其进行处理。

3. 软中断

软中断的概念主要来源于UNIX系统。软中断是对应于硬中断而言的。通过硬件产生相应的中断请求，称为硬中断。而软中断则不然，它是在通信进程之间通过模拟硬中断而实现的一种通信方式。中断源发出软中断信号后，CPU或者接收进程在“适当的时机”进行中断处理或者完成软中断信号所对应的功能。这里“适当的时机”，表示接收软中断信号的进程须等到该接收进程得到处理器之后才能进行。如果该接收进程是占据处理器的，那么，该接收进程在接收到软中断信号后将立即转去执行该软中断信号所对应的功能。

4. 中断处理过程

一旦CPU响应中断，转入中断处理程序，系统就开始进行中断处理。下面对中断处理过程进行详细说明：

- 1) CPU检查响应中断的条件是否满足。CPU响应中断的条件是：有来自于中断源的中断请求、CPU允许中断。如果中断响应条件不满足，则中断处理无法进行。
- 2) 如果CPU响应中断，则CPU关中断，使其进入不可再次响应中断的状态。
- 3) 保存被中断进程现场。为了在中断处理结束后能使进程正确地返回到中断点，系统必须保存当前处理器状态字PSW和程序计数器PC等的值。这些值一般保存在特定堆栈或硬件寄存器中。
- 4) 分析中断原因，调用中断处理子程序。在多个中断请求同时发生时，处理优先级最高的中断源发出的中断请求。在系统中，为了处理上的方便，通常都是针对不同的中断源编制有不同的中断处理子程序（陷阱处理子程序）。这些子程序的入口地址（或陷阱指令的入口地址）存放在内存的特定单元中。再者，不同的中断源也对应着不同的处理器状态字PSW。这些不同的PSW被放在相应的内存单元中，与中断处理子程序入口地址一起构成中断向量。显然，根据中断或陷阱的种类，系统可由中断向量表迅速地找到该中断响应的优先级、中断处理子程序（或陷阱指令）的入口地址和对应的PSW。
- 5) 执行中断处理子程序。对陷阱来说，在有些系统中则是通过陷阱指令向当前执行进程发出软中断信号后调用对应的处理子程序执行。
- 6) 退出中断，恢复被中断进程的现场或调度新进程占据处理器。
- 7) 开中断，CPU继续执行。

5. 设备管理程序与中断方式

处理器的高速和输入输出设备低速之间的矛盾，是设备管理要解决的一个重要问题。为了提高整体效率，减少在程序直接控制方式中的CPU等待时间以及提高系统的并行工作效率，采用中断方式来控制输入输出设备和内存与CPU之间的数据传送，是很有必要的。

在硬件结构上，这种方式要求CPU与输入输出设备（或控制器）之间有相应的中断请求线，而且在输入输出设备控制器的控制状态寄存器上有相应的中断允许位。

在中断方式下，中央处理器与I/O设备之间数据传输的大致步骤如下。

1) 首先，某个进程需要数据时，发出指令启动输入输出设备准备数据。同时该指令还通知输入输出设备控制状态寄存器中的中断允许位打开，以便在需要时，中断程序可以被调用执行。

2) 在进程发出指令启动设备之后，该进程放弃处理器，等待相关I/O操作完成。此时，进程调度程序会调度其他就绪进程使用处理器。另一种方式是该进程在能够运行的情况下将继续运行，直到中断信号来临。

3) 当I/O操作完成时，输入输出设备控制器通过中断请求线向处理器发出中断信号。处理器收到中断信号之后，转向预先设计好的中断处理程序对数据传送工作进行相应的处理。

4) 得到了数据的进程，转入就绪状态。在随后的某个时刻，进程调度程序会选中该进程继续工作。

显然，当处理器发出启动设备和允许中断指令之后，处理器已被调度程序分配给其他进程。也可以启动不同的设备和允许中断指令，从而做到设备与设备间的并行操作以及设备和处理器间的并行操作。中断方式使处理器的利用率提高且能支持多道程序和设备的并行操作。

但是中断方式仍然存在一些问题。首先，在I/O控制器的数据缓冲寄存器装满数据之后将会发生中断。如果数据缓冲寄存器比较小，那么，在数据传送过程中，发生中断次数较多。这将耗去大量的CPU处理时间。

其次，现代计算机系统通常配置有各种各样的输入输出设备。如果这些输入输出设备都通过中断处理方式进行并行操作，那么中断次数的急剧增加会造成CPU无法响应中断和出现数据丢失现象。

6.2.3 设备驱动程序

设备驱动程序是直接同硬件打交道的软件模块。一般而言，设备驱动程序的任务是接受来自与设备无关的上层软件的抽象请求，进行与设备相关的处理。

1. 设备驱动程序的功能

设备驱动程序主要有以下四个方面的处理工作：

- 向有关的输入输出设备的各种控制器（的寄存器）发出控制命令，并且监督它们的正确执行，进行必要的错误处理。
- 对各种可能的有关设备排队、挂起、唤醒等操作进行处理。
- 执行确定的缓冲区策略。
- 进行比寄存器接口级别层次更高的一些特殊处理，如代码转换、ESC处理等。它们均是依赖于设备的，所以不适合放在高层次的软件中处理。

2. 设备驱动程序的特性

设备驱动程序的最突出的特点是，它与I/O设备的硬件结构密切联系。设备驱动程序中全部是依赖于设备的代码。设备驱动程序是操作系统底层中唯一知道各种输入输出设备的控制器细节及其用途的部分。

例如，只有磁盘驱动程序具体了解磁盘的区段、磁道、柱面、磁头、臂的运动、交错访问系

数、马达驱动器、磁头定位次数，以及所有保证磁盘正常工作的机制，其他软件根本不过问这些硬件操作细节。

3. 设备驱动程序的结构

首先，不同的操作系统中，对设备驱动程序的结构的要求是不同的。一般而言，在操作系统的相关文档中，都有对设备驱动程序结构方面的统一要求。

其次，设备驱动程序的结构同输入输出设备的硬件特性有关。一台彩色显示器的设备驱动程序的结构，显然同磁盘设备驱动程序的结构不同。通常一个设备驱动程序对应处理一种设备类型，或者至多一类密切联系着的设备。系统往往对略有差异的一类设备提供一个通用的设备驱动程序。例如，在Microsoft Windows 9x中，为CD-ROM提供一个通用的设备驱动程序。对不同品牌或不同性能的IDE CD-ROM，用户都可以用这个CD-ROM设备驱动程序。但是，为了追求更好的性能，用户往往放弃使用这个通用的设备驱动程序，而使用厂家提供的专为一种CD-ROM编写的设备驱动程序。

可见对于某一类设备而言，是采用通用的设备驱动程序，还是采用专用的设备驱动程序，取决于用户在这台输入输出设备上追求的目标。如果把设备安装的便利性放在第一位，那么建议考虑使用该类设备的通用驱动程序；如果优先考虑设备的运行效率，那么当然应该首选专门为这台设备编写的驱动程序。

4. 设备驱动程序层的内部策略

设备驱动程序层的内部策略包括以下几方面。

1) 确定是否发请求：典型的请求是读磁盘第 n 块数据。如果驱动程序在一个请求到来时空闲，它就立即开始实施该请求；然而，倘若它已经忙于应付另一个请求，则通常就把这个新的请求排进请求队列，尽快予以处理。

2) 确定发什么：譬如说对于磁盘，实际应答I/O请求的第一步，是把它的用语从抽象向具体转换。对一个磁盘驱动程序来说，这意味着弄清楚被请求的块在磁盘上的实际位置，检查驱动器的马达是否在运转，确定臂是否放在恰当的柱面上，诸如此类。总而言之，它必须决定需要控制器的哪些操作，以及按照什么样的次序。

3) 发布命令：一旦明确向控制器发布哪些命令，设备驱动程序就通过写入该控制器设备寄存器，着手把命令发出去。有些控制器一次只能处理一条命令；另一些控制器则可接受一张命令链接表，然后自行执行所有命令，不用再求助于操作系统。

4) 发后处理：在一条或多条指令发出以后，存在着两种做法。在多数情况下，设备驱动程序必须等待控制器为它扫清道路，所以它本身阻塞，直至中断来把它唤醒；在其他情况下，操作毫不拖延地完成，所以驱动程序无需阻塞。一个例子是，滚动某些终端的屏幕只消把几个字节写入控制器的寄存器即可，毋须任何机械的运动，整个操作可以在几个微秒中完成。第二个例子是采用缓冲的输出过程，只须向缓冲区写入即可返回，无需阻塞，真正的输出由中断处理程序完成。在操作完成之后，不管哪种做法都必须检查错误。只不过对于阻塞的情况，将在中断处理中检查，对于不阻塞的情况，在此处马上检查。

5) 中断时被调用的驱动程序的事后处理：

- 检查结果状态和传送结果数据：如果正确，驱动程序可令数据流向与设备无关的软件（例如刚读过的一块）。
- 可能的错误处理：它返回一些错误状态信息，汇报给它的调用者。
- 可能的唤醒：即如果有因等待此操作完成而阻塞的进程，则唤醒之。
- 可能启动下一个I/O操作，或者因无请求而阻塞。倘若有其他请求在排队，现在即可挑选其一加以启动；如果没有，该驱动程序则阻塞起来，等候下一请求的到来。

6.2.4 与设备无关的系统软件

除了一些I/O软件与设备相关之外，大部分软件是与设备无关的。至于设备驱动程序与设备无关的软件之间的界限如何划分，则随操作系统的不同而不同。具体划分原则取决于系统的设计者怎样权衡系统与设备的独立性、驱动程序的运行效率等诸多因素。对于一些按照设备独立方式实现的功能，出于效率和其他方面的考虑，也可以由设备驱动程序实现。图6-2给出了常见的设备无关软件层实现的一些功能。

一般而言，所有设备都需要的I/O功能可以在与设备独立的软件中实现。这类软件面向应用层并提供一个统一的接口。

设备驱动程序的统一接口
设备命名
设备保护
提供一个与设备无关的逻辑块
缓冲
存储设备的块分配
独占设备的分配和释放
错误处理

图6-2 与设备无关I/O软件的功能

1. 统一命名

我们曾经说过，在操作系统的I/O软件中，对输入输出设备采用了统一命名。那么，谁来区分这些命名同文件一样的输入输出设备呢？这就是与设备无关的软件，它负责把设备的符号名映射到相应的设备驱动程序上。

举例来说，在UNIX系统中，像 / dev / tty00这样的设备名，唯一确定了一个特殊文件的i节点，这个i节点包含了主设备号和从设备号。主设备号用于寻找对应的设备驱动程序，而从设备号提供了设备驱动程序的有关参数，用来确定要读写的具体设备。

2. 设备保护

对设备进行必要的保护，防止无授权的应用或用户的非法使用，是设备保护的主要作用。

设备保护是与设备命名的机制密切相关的。那么，在操作系统中如何防止无授权的用户存取设备呢？这也取决于具体的系统，比如在MS - DOS中，操作系统根本没有对设备设计任何的保护机制。不过在大型的计算机系统中，用户进程对I/O设备的直接访问是完全禁止的。而UNIX系统则采用一种存取权限的模式，对于系统中的I/O设备，这类特殊文件提供“rwx”位进行保护，系统管理员可以根据需要为每一个设备设置适当的存取权限。

3. 提供与设备无关的逻辑块

在各种输入输出设备中，有着不同的存储设备，其空间大小、读取速度和传输速率等各不相同。比如，当前台式机和服务器中常用的硬盘，其空间大小在若干个吉字节。而在掌上电脑和数码相机这一类设备中，则使用闪存这种存储器，其容量一般在数十兆字节。又如，目前高档的打印机都自带缓冲存储器，它们可能是一个硬盘，也可能是随机存储芯片或者闪存。它们的空间大

小、读取速度和传输速率都极不相同。因此，与设备无关的软件就有必要向较高层软件屏蔽各种I/O设备空间大小、处理速度和传输速率各不相同的这一事实，而向上层提供大小统一的逻辑块尺寸。

这样，较高层的软件只与抽象设备打交道，不考虑物理设备空间和数据块大小而使用等长的逻辑块。这些差别在这一层都隐藏起来了。

4. 缓冲

对于常见的块设备和字符设备，一般都使用缓冲区。对块设备，硬件一般一次读写一个完整的块，而用户进程是按任意单位读写数据的。如果用户进程只写了半块数据，则操作系统通常将数据保存在内部缓冲区，等到用户进程写完整块数据才将缓冲区的数据写到磁盘上。对字符设备，当用户进程把数据写给系统的速度快于系统输出数据速率时，也必须使用缓冲。

5. 存储设备的块分配

在创建一个文件并向其中填入数据时，通常在硬盘中要为该文件分配新的存储块。为完成这一分配工作，操作系统需要为每个磁盘设置一张空闲块表或位图，这种查找一个空闲块的算法是与设备无关的，因此可以放在设备驱动程序上面与设备无关的软件层中处理。

6. 独占设备的分配和释放

有一些设备，如打印机驱动器，在任一时刻只能被单个进程使用。这就要求操作系统对设备使用请求进行检查，并根据申请设备的可用状况决定是接收该请求还是拒绝该请求。一个简单的处理这些请求的方法是，要求进程直接通过OPEN打开设备的特殊文件来提出请求。若设备不能用，则OPEN失败。关闭这种独占设备的同时释放该设备。

7. 出错处理

一般来说，出错处理是由设备驱动程序完成的。大多数错误是与设备密切相关的，因此，只有驱动程序知道应如何处理（比如，重试、忽略或放弃）。但还有一些典型的错误不是输入输出设备的错误造成的，如由于磁盘块受损而不能再读，驱动程序将尝试重读一定次数。若仍有错误，则放弃重读并通知与设备无关的软件，这样，如何处理这个错误就与设备无关了。如果在读一个用户文件时出现错误，操作系统会将错误信息报告给调用者。若在读一些关键的系统数据结构时出现错误，比如磁盘的空闲块位图，操作系统则需打印错误信息，并向系统管理员报告相应错误。

6.2.5 用户空间的I/O软件

一般来说，大部分I/O软件都包含在操作系统中，但是用户程序仍有一小部分是与库函数连接在一起的，甚至还有在内核之外运行的程序。通常的系统调用，包括I/O系统调用，是由库函数实现。如，一个用C语言编写的程序可含有如下的系统调用：

```
count = write (fd, buffer, nbytes);
```

在这个程序运行期间，该程序将与库函数write连接在一起，并包含在运行时的二进制程序代码中。显然，所有这些库函数是设备管理I/O系统的组成部分。通常这些库函数所做的工作主要是把系统调用时所用的参数放在合适的位置，由其他的I/O过程去实现真正的操作。在这里，

输入输出的格式是由库函数完成的。标准的I/O库包含了许多涉及I/O的过程，它们都是作为用户程序的一部分运行的。

下面以C语言中的printf为例说明。printf以一个格式串和可能的一些变量作为输入，构造一个ASCII字符串，然后调用WRITE这个系统调用输出这个串。对输入而言，类似的过程是gets，它读入一行并返回一个字符串。

但是，并非所有的用户层I/O软件都是由库函数组成的。Spooling（假脱机）系统是另一种重要的处理方法。Spooling系统是多程序设计系统中处理独占I/O设备的一种方法。

假设有一种典型的假脱机设备——行式打印机，一个进程打开了它，然后很长时间不使用，这样就导致了其他进程都无法使用这台打印机打印。

解决方法是创建一个特殊进程，称为守护(daemon)进程，以及一个特殊目录，称为spooling目录。当一个进程要打印一个文件时，首先要生成打印的整个文件，将其放在spooling目录下。然后由守护进程完成该目录下文件的打印工作，该进程是唯一一个拥有使用打印机特殊文件权限的进程。而且，通过保护特殊文件以防止用户直接使用，可以解决进程空占打印机的问题。

需要指出的是，Spooling技术不仅仅只适用于打印机这类输入输出设备，还可应用到其他一些情况。例如，在Internet上的USENET电子邮件系统中，成千上万台计算机联在一起。如果要通过USENET向某人发送邮件，先调用一个称为send的程序，send接到要发出的信件，然后将它送入一个spooling目录，待以后发送。整个邮件系统是运行在操作系统之外的。

图6-3总结了I/O软件的所有层次及每一层的主要功能。

图中的箭头给出了I/O部分的控制流。这里我们举一个读硬盘文件的例子。当用户程序试图读一个硬盘文件时，需要通过操作系统实现这一操作。与设备无关软件检查高速缓存中是否有要读的数据块。若没有，则调用设备驱动程序，向I/O硬件发出一个请求。然后，用户进程阻塞并等待磁盘操作的完成。当磁盘操作完成时，硬件产生一个中断，转入中断处理程序。中断处理程序检查中断的原因，认识到这是磁盘读取操作已经完成，于是唤醒用户进程取回从磁盘读取的信息，从而结束了此次I/O请求。用户进程在得到了所需的硬盘文件内容之后，继续运行。

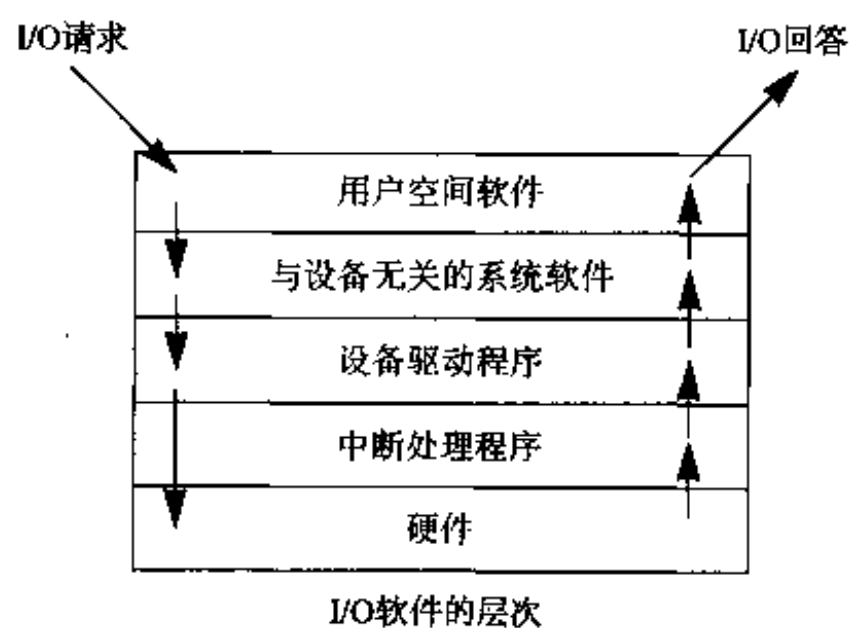


图6-3 I/O系统的层次结构及每层的主要功能

6.3 Windows 2000/XP的I/O系统结构和模型

Microsoft Windows 2000/XP I/O系统是Windows 2000/XP执行体的组件，存在于NTOSKRNL.EXE文件中。它接受I/O请求（来自用户态和核心态的调用程序），并且以不同的形式把它们传送到I/O设备。在用户态I/O函数和实际的I/O硬件之间有几个分立的系统组件，包括文件系统驱动程序、过滤器驱动程序和低层设备驱动程序。

Windows 2000/XP I/O系统的设计目标如下：

- 在单处理器或多处理器系统中都可以快速进行I/O处理。
- 使用标准的Windows 2000/XP安全机制保护共享的资源。
- 满足Microsoft Win32、OS/2和POSIX子系统指定的I/O服务的需要。
- 提供服务，使设备驱动程序的开发尽可能地简单，并且允许用高级语言编写驱动程序。
- 根据用户的配置或者系统中硬件设备的添加和删除，允许在系统中动态地添加或删除相应的设备驱动程序。
- 通过添加驱动程序透明地修改其他驱动程序或设备的行为。
- 为包括FAT、CD-ROM文件系统（CDFS）、UDF（Universal Disk Format，统一磁盘格式）文件系统和Windows 2000/XP文件系统（NTFS）的多种可安排的文件系统提供支持。
- 允许整个系统或者单个硬件设备进入和离开低功耗状态，这样可以节约能源。

在本节以后的部分中首先介绍Windows 2000/XP I/O系统的结构和组件以及不同类型的设备驱动程序，然后看一下描述设备、设备驱动程序和I/O请求的关键的数据结构以及设备驱动程序分类和结构。最后，将进一步分析在整个系统中完成I/O请求的必要步骤。

Windows 2000/XP I/O系统由一些执行体组件和设备驱动程序组成，如图6-4所示。

- I/O管理器把应用程序和系统组件连接到各种虚拟的、逻辑的和物理的设备上，并且定义了一个支持设备驱动程序的基本构架。
- 设备驱动程序为某种类型的设备提供一个I/O接口。设备驱动程序从I/O管理器接受处理命令，当处理完毕后通知I/O管理器。设备驱动程序之间的协同工作也通过I/O管理器进行。
- PnP（即插即用，plug and play）管理器通过与I/O管理器和总线驱动程序的协同工作来检测硬件资源的分配，并且检测相应硬件设备的添加和删除。
- 电源管理器通过与I/O管理器的协同工作来检测整个系统和单个硬件设备，完成不同电源状态的转换。
- WMI（Windows Management Instrumentation）支持例程，也叫做Windows驱动程序模型（WDM，Windows Driver Model）WMI提供者，允许驱动程序使用这些支持例程作为媒介，与用户态运行的WMI服务通信。
- 注册表作为一个数据库，存储基本硬件设备的描述信息以及驱动程序的初始化和配置信息。
- 硬件抽象层（HAL）I/O访问例程把设备驱动程序与多种多样的硬件平台隔离开来，使它们在给定的体系结构中是二进制可移植的，并在Windows 2000/XP支持的硬件体系结构中是源代码可移植的。

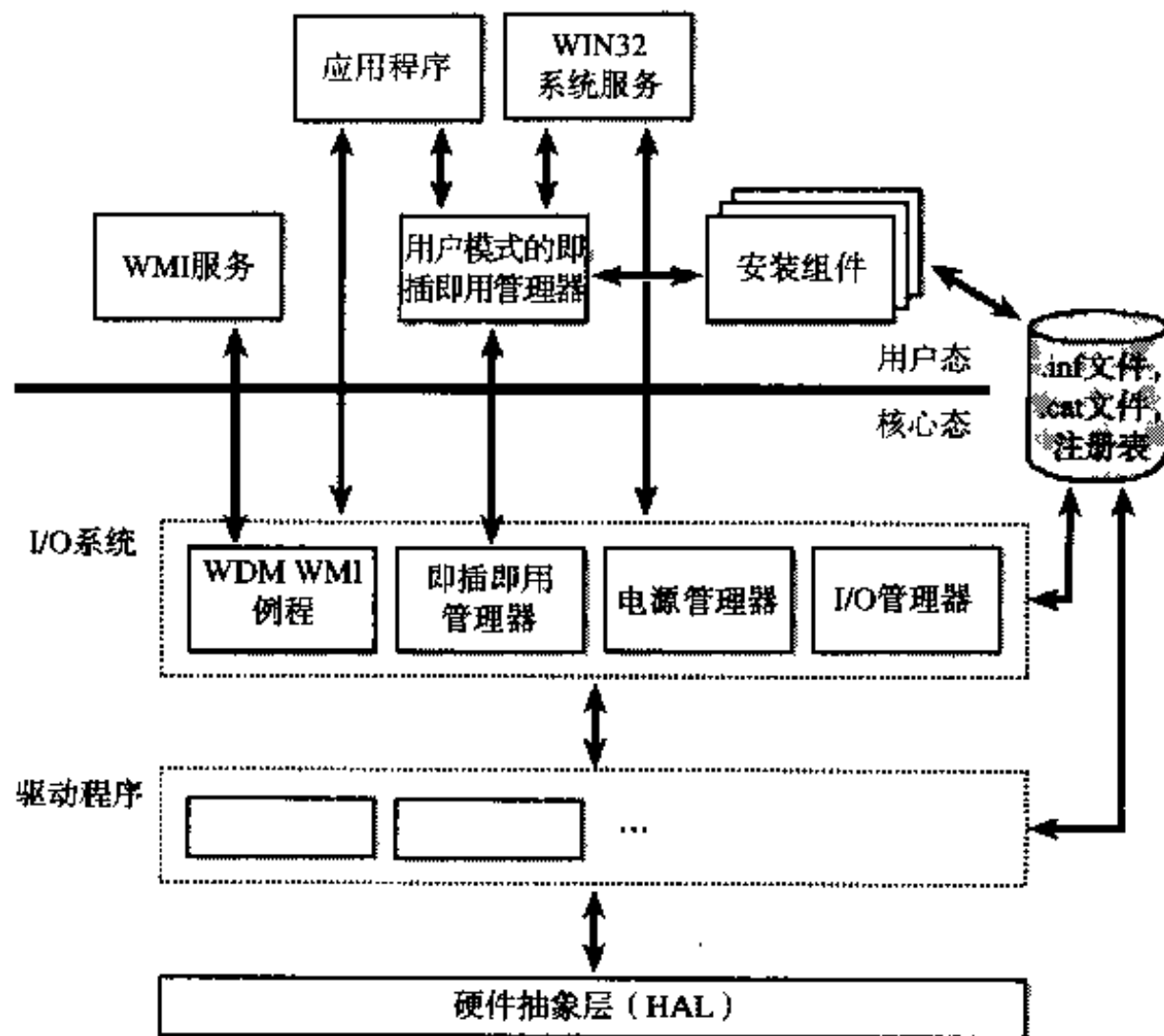


图6-4 I/O系统组件

大部分I/O操作并不会涉及所有的I/O组件，一个典型的I/O操作从应用程序调用一个与I/O操作有关的函数（例如从一个设备中读取数据）开始，通常会涉及I/O管理器、一个或多个设备驱动程序以及硬件抽象层。

在Windows 2000/XP中，所有的I/O操作都通过虚拟文件执行，隐藏了I/O操作目标的实现细节，为应用程序提供了一个统一的到设备的接口界面。虚拟文件是指用于I/O的所有源或目标，它们都被当做文件来处理（例如文件、目录、管道和邮箱）。所有被读取或写入的数据都可以看作是直接读写到这些虚拟文件的流。用户态应用程序（不管它们是Win32、POSIX或OS/2）调用文档化的函数，这些函数再依次调用内部I/O子系统函数来从文件中读取、对文件写入和执行其他的操作。I/O管理器动态地把这些虚拟文件请求指向适当的设备驱动程序。一个典型的I/O请求流程的结构如图6-5所示。

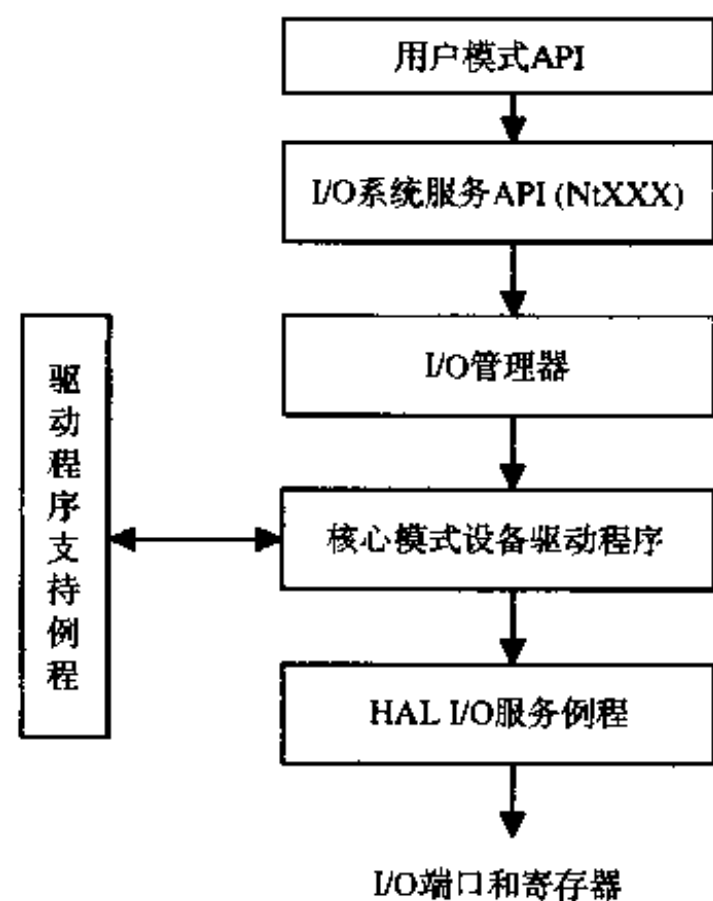


图6-5 一个典型的I/O请求流程

下面将更进一步讨论其中的一些组件，更详细地叙述I/O管理器，论述不同类型的设备驱动

程序和关键的I/O系统数据结构，并介绍PnP管理器和电源管理器的结构。

6.3.1 I/O管理器

“I/O管理器”(I/O manager)定义有序的工作框架(或模型)，在该框架里，I/O请求被提交给设备驱动程序。在Windows2000/XP中，整个I/O系统是由“包”驱动的，大多数I/O请求用“I/O请求包(IRP)”表示，它从一个I/O系统组件移动到另一个I/O系统组件(快速I/O是一个特例，它不使用IRP)。IRP是在每个阶段控制如何处理I/O操作的数据结构(关于IRP的详细信息，请参阅6.4.3节“I/O请求包”)。

I/O管理器创建代表每个I/O操作的IRP，传递IRP给正确的驱动程序，并且当此I/O操作完成后，处理这个数据包。相反，驱动程序接受IRP，执行IRP指定的操作，并且在完成操作后把IRP送回I/O管理器或为下一步的处理而通过I/O管理器把它送到另一个驱动程序。

除了创建并处理IRP以外，I/O管理器还为不同的驱动程序提供了公共的代码，驱动程序调用这些代码来执行它们的I/O处理。通过在I/O管理器中合并公共的任务，单个的驱动程序将变得更加简洁和更加紧凑。例如，I/O管理器提供一个允许某个驱动程序调用其他驱动程序的函数。它还管理用于I/O请求的缓冲区，为驱动程序提供超时支持，并记录操作系统中加载了哪些可安装的文件系统。这些支持例程已包含在DDK中。

I/O管理器也提供灵活的I/O服务，允许环境子系统(例如，Win32和POSIX)执行它们各自的I/O函数。这些服务包括用于异步I/O的高级服务，它们允许开发者建立可升级的高性能的服务器应用程序。

驱动程序呈现的统一的、模块化的接口允许I/O管理器调用任何驱动程序而不需要与它的结构和内部细节有关的任何特殊的知识。驱动程序也可以相互调用(通过I/O管理器)来完成I/O请求的分层的、独立的处理。

6.3.2 PnP管理器

即插即用(Plug and Play, PnP)是计算机系统I/O设备与部件配置的应用技术。顾名思义，PnP是指插入就可用，不需要进行任何设置操作。

由于一个系统可以配置多种外部设备，设备也经常变动和更换，它们都要占有一定的系统资源，彼此间在硬件和软件上可能会产生冲突。因此在系统中要正确地对它们进行配置和资源匹配；当设备撤除、添置和进行系统升级时，配置过程往往是一个困难的过程。为了改变这种状况，出现了PnP技术。

PnP技术主要有以下特点：PnP技术支持I/O设备及部件的自动配置，使用户能够简单方便地使用系统扩充设备；PnP技术减少了由制造商造成的种种用户限制，简化了部件的硬件跳线设置，使I/O附加卡和部件不再具有人工跳线设置电路；利用PnP技术可以在主机板和附加卡上保存系统资源的配置参数和分配状态，有利于系统对整个I/O资源的分配和控制；PnP技术支持和兼容各种操作系统平台，具有很强的扩展性和可移植性；PnP技术在一定程度上具有“热插入”、“热拼接”功能。

PnP技术的实现需要多方面的支持，其中包括具有PnP功能的操作系统、配置管理软件、软

件安装程序和设备驱动程序等；另外还需要系统平台的支持（如PnP主机板、控制芯片组和支持PnP的BIOS等）以及各种支持PnP规范的总线、I/O控制卡和部件。

PnP管理器为Windows 2000/XP提供了识别并适应计算机系统硬件配置变化的能力。PnP支持需要硬件、设备驱动程序和操作系统的协同工作才能实现。关于总线上设备标识的工业标准是实现PnP支持的基础，例如，USB标准定义了USB总线上识别USB设备的方式。Windows 2000/XP的PnP支持提供了以下能力：

- PnP管理器自动识别所有已经安装的硬件设备。在系统启动的时候，一个进程会检测系统中硬件设备的添加或删除。
- PnP管理器通过一个名为资源仲裁（resource arbitrating）的进程收集硬件资源需求（中断，I/O地址等）来实现硬件资源的优化分配，满足系统中的每一个硬件设备的资源需求。PnP管理器还可以在启动后根据系统中硬件配置的变化对硬件资源重新进行分配。
- PnP管理器通过硬件标识选择应该加载的设备驱动程序。如果找到相应的设备驱动程序，则通过I/O管理器加载，否则启动相应的用户态进程请求用户指定相应的设备驱动程序。
- PnP管理器也为检测硬件配置变化提供了应用程序和驱动程序的接口。因此在Windows 2000/XP中，在硬件配置发生变化时，相应的应用程序和驱动程序也会得到通知。

Windows 2000/XP的目标是提供完全的PnP支持，但是具体的PnP支持程度要由硬件设备和相应驱动程序共同决定。如果某个硬件或驱动程序不支持PnP，整个系统的PnP支持将受到影响。一个不支持PnP的驱动程序可能会影响其他设备的正常使用。一些比较早的设备和相应的驱动程序可能都不支持PnP。在NT4下可以正常工作的驱动程序一般情况下在Windows 2000/XP中也可以工作，PnP就不能通过这些驱动程序完成设备资源的动态配置。

为了支持PnP，设备驱动程序必须支持PnP调度（dispatch）例程和添加设备的例程，总线驱动程序必须支持不同类型的PnP请求。在系统启动的过程中，PnP管理器向总线驱动程序询问得到不同设备的描述信息，包括设备标识、资源分配需求等，然后PnP管理器就加载相应的设备驱动程序并调用每一个设备驱动程序的添加设备例程。

设备驱动程序加载后已经做好了开始管理硬件设备的准备，但是并没有真正开始和硬件设备通信。设备驱动程序等待PnP管理器向其PnP调度例程发出启动设备（start-device）的命令，启动设备命令中包含PnP管理器在资源仲裁后确定的设备的硬件资源分配信息。设备驱动程序收到启动设备命令后开始驱动相应设备并使用所分配的硬件资源开始工作。

设备启动后，PnP管理器可以向设备驱动程序发送其他的PnP命令，包括把设备从系统中卸载，重新分配硬件资源等。把设备从系统中移开包括的PnP命令有query-remove, remove等，重新分配硬件资源涉及的PnP命令有query-stop, stop, start-device等。不同的PnP命令会引起设备状态的改变，如图6-6所示。

6.3.3 电源管理器

同Windows 2000/XP的PnP支持一样，电源管理也需要底层硬件的支持，底层的硬件需要符合ACPI（Advanced Configuration and Power Interface）标准。因此支持电源管理的计算机系统的

BIOS (Basic Input and Output System) 必须符合ACPI标准, 1998年底以来的x86计算机系统都符合ACPI标准。

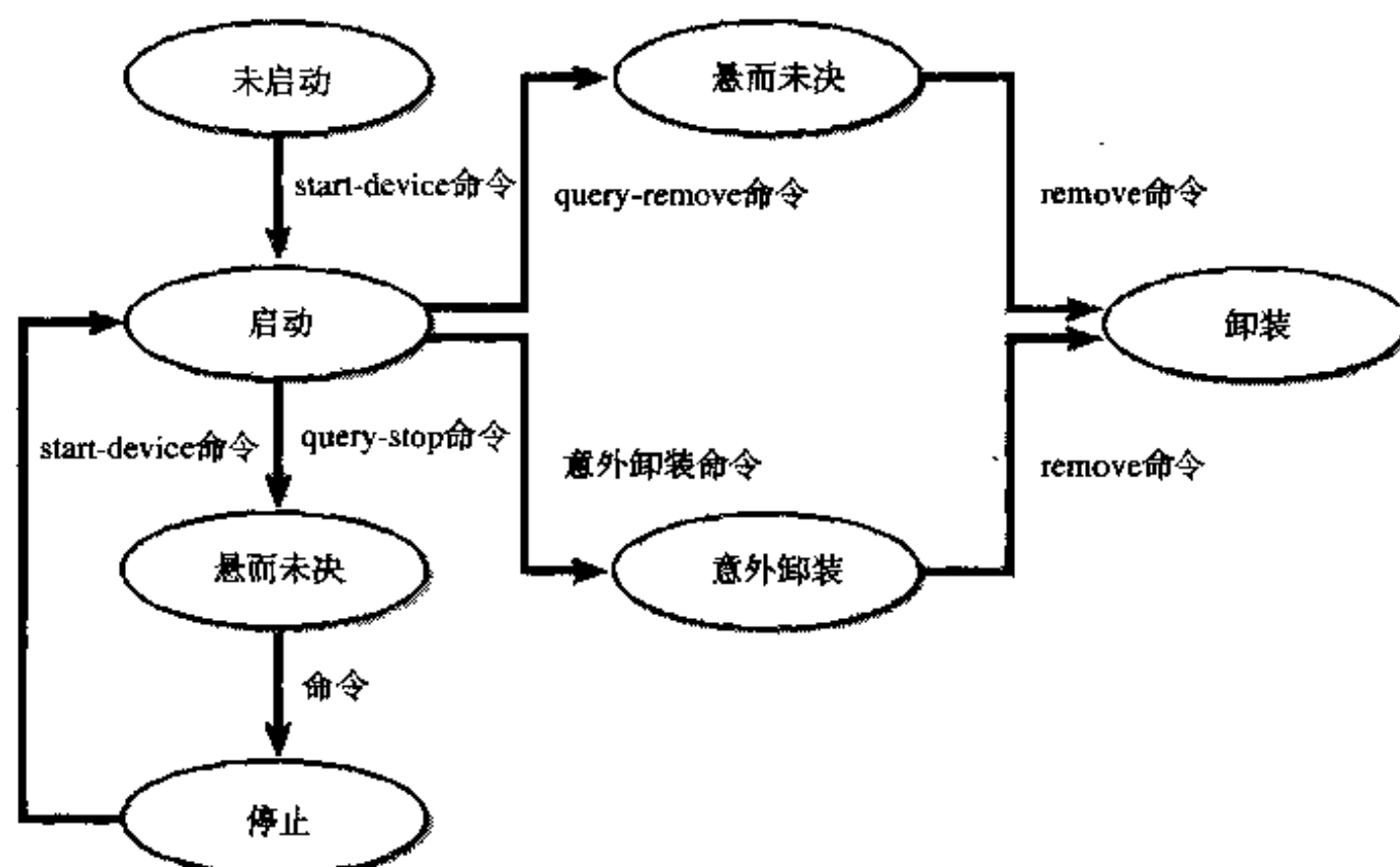


图6-6 PnP状态转换

ACPI为系统和设备定义了不同的能耗状态, 目前共有六种, 从S0 (正常工作) 到S5 (完全关闭), 如表6-1所示。每一种状态都有如下指标:

- 电源消耗: 计算机系统消耗的能源。
- 软件运行恢复: 计算机系统回复到正常工作状态时软件能否恢复运行。
- 硬件延迟: 计算机系统回复到正常工作状态的时间延迟。

表6-1 系统能耗状态定义

状 态	能 耗	软件恢复	硬件延迟
S0 (正常工作)	最大	无	无
S1 (睡眠)	比S0大, 比S2小	恢复运行	小于2秒
S2 (睡眠)	比S1大, 比S3小	恢复运行	2秒或更多
S3 (睡眠)	比S2大, 比S4小	恢复运行	2秒或更多
S4 (休眠)	电源按钮保持微弱电流, 系统保持唤醒电流	恢复运行	长
S5 (完全关闭)	电源按钮一直保持微弱电流	系统引导	长

计算机系统在从S1到S4的状态之间互相转换, 转换必须先通过状态S0。如图6-7所示, S1 ~ S5的状态转换到S0称作唤醒, 从S0转换到S1 ~ S5称作睡眠。

设备也有相应的能耗状态, 设备能耗状态的分类和整个计算机系统是不同的。ACPI定义的设备能耗分为四种状态: 从D0到D3。其中D0为正常工作, D3为关闭, D1和D2的意义可以由设备和驱动程序自行定义, 只要保证D1耗能低于D2, D2耗能低于D3即可。表6-2是一个系统到设

备能耗状态映射的例子。

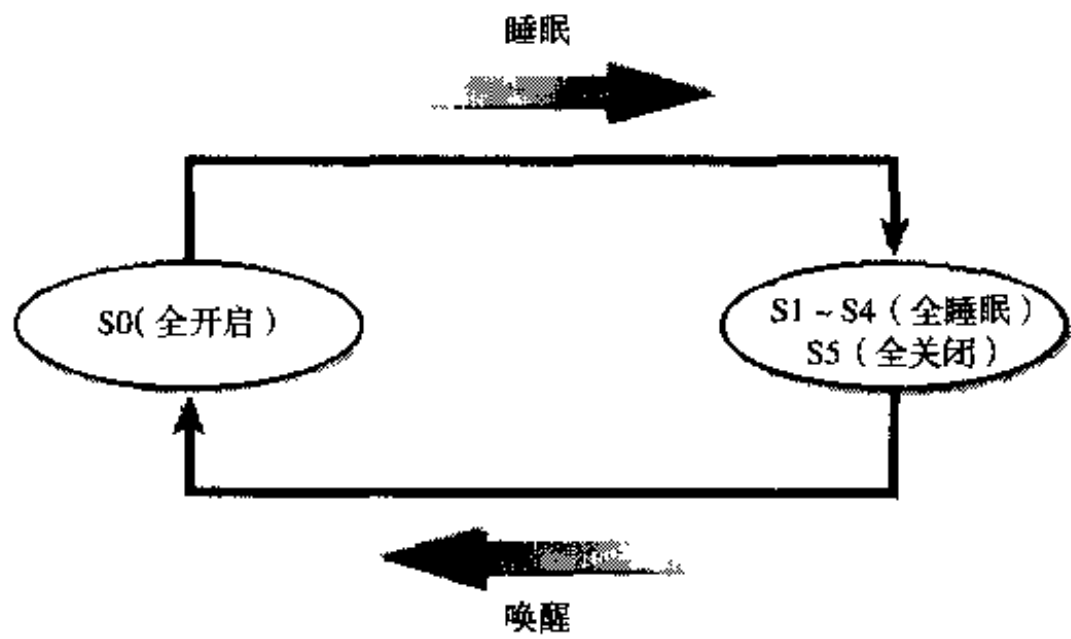


图6-7 电源状态转换

表6-2 系统到设备能耗状态映射的例子

系统能耗状态	设备能耗状态
S0	D0 (正常工作)
S1	D2
S2	D2
S3	D2
S4	D3 (关闭)
S5	D3 (关闭)

Windows 2000/XP的电源管理策略由两部分组成：电源管理器和设备驱动程序。电源管理器是系统电源策略的所有者，因此整个系统的能耗状态转换由电源管理器决定，并调用相应设备的驱动程序完成。电源管理器根据以下因素决定当前相同的能耗状态：

- 系统活动状况
- 系统电源状况
- 应用程序的关机、休眠请求
- 用户的操作，例如用户按电源按钮
- 控制面板的电源设置

当电源管理器决定要转换能耗状态时，相应的电源管理命令会发给设备驱动程序的相应调度例程。一个设备可能需要多个设备驱动程序，但是负责电源管理的设备驱动程序只有一个，设备驱动程序根据当前系统状态和设备的状态决定如何进行下一步操作，例如，当设备状态从S0切换到S1时，设备的能耗状态也从D0切换到D1。

除了响应电源管理器的电源管理命令外，设备驱动程序也可以独立地控制设备的能耗状态。在一些情况下，当设备长时间不用时，设备驱动程序就可以减小该设备的能耗。设备驱动程序可以自己检测设备的闲置时间，也可以通过电源管理器检测。在使用电源管理器时，设备驱动程序

通过调用函数PoRegisterDeviceForIdleDetection将相应设备注册到电源管理器中，该函数告诉电源管理器检测设备闲置的超时参数以及发现设备闲置时应该把设备切换到何种能耗状态。驱动程序需要设置两个超时值，一个用于配置计算机节省电能，另一个用于计算机达到最优性能。调用了PoRegisterDeviceForIdleDetection函数后，设备驱动程序需要通过函数PoSetDeviceBusy通知电源管理器设备何时被激活。

6.4 Windows 2000/XP I/O系统的数据结构

四种主要的数据结构代表了I/O请求：文件对象、驱动程序对象、设备对象和I/O请求包 (IRP)。

6.4.1 文件对象

文件明显符合Windows 2000/XP中的对象标准：它们是两个或两上以上用户态进程的线程可以共享的系统资源；它们可以有名称；它们被基于对象的安全性所保护；并且它们支持同步。虽然在Windows 2000/XP中的大多数共享资源是基于内存的资源，但是I/O系统管理的大多数资源位于物理设备中或者就是实际的物理设备。尽管这有些差异，但在I/O系统中的共享资源，像在Windows 2000/XP执行体的其他组件中的一样，都作为对象而被操作。

文件对象提供了基于内存的共享物理资源的表示法（除了被命名的管道和邮箱以外，它们虽然是基于内存的，但不是物理资源）。在Windows 2000/XP I/O系统中，文件对象也代表这些资源。表6-3列出了一些文件对象的属性。要得到特殊字段的声明和大小，请参阅NTDDK.H中FILE_OBJECT的结构定义。

表6-3 文件对象属性

属 性	目 的
文件名	标识文件对象指向的物理文件
字节偏移量	在文件中标识当前位置（只对同步I/O有效）
共享模式	表示当调用者正在使用文件时，其他的调用者是否可以打开文件来做读取、写入或删除操作
打开模式	表示I/O是否将被同步或异步、高速缓存或不高速缓存、连续或随机等
指向设备对象的指针	表示文件在其上驻留的设备的类型
指向卷参数块的指针	表示文件在其上驻留的卷或分区
指向区域对象指针的指针	表示描述一个映射文件的根结构
指向专用高速缓存映射的指针	表示文件的哪一部分由高速缓存管理器管理，以及它们驻留在高速缓存的什么地方

当调用者打开文件或单一的设备时，I/O管理器将为文件对象返回一个句柄。图6-8说明了一个文件被打开时所发生的情况。

在这个例子中，C程序调用库函数fopen，由它去调用Win32的CreateFile函数。然后Win32子

系统DLL（在这里是KERNEL32.DLL）在NTDLL.DLL中调用本地NtCreateFile函数，在NTDLL.DLL中的例程包含引发到核心态系统服务调度程序转换的适当指令。最后，系统服务调度程序在NTOSKRNL.EXE中调用真正的NtCreateFile例程。

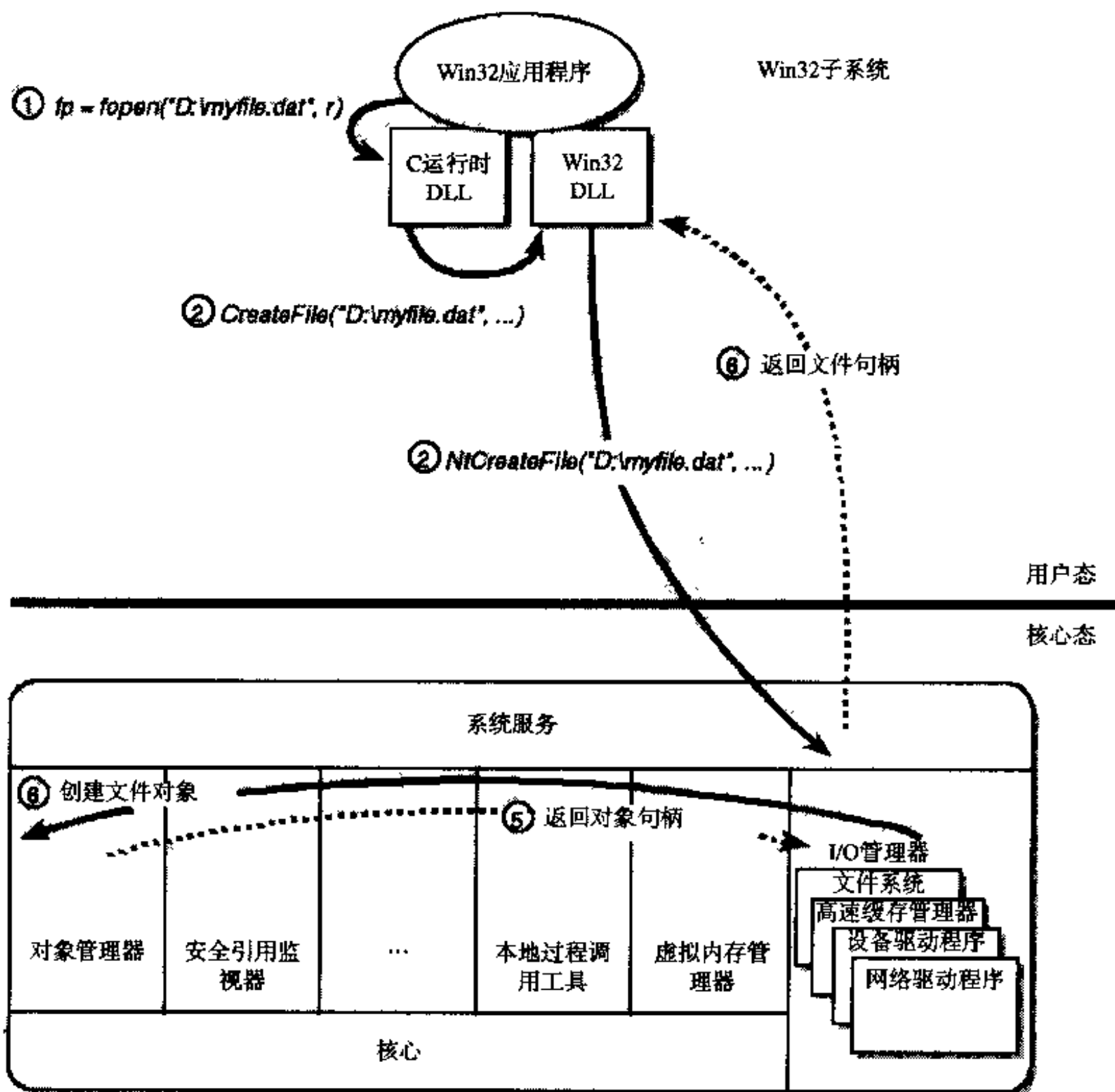


图6-8 打开一个文件对象

像其他的执行体对象一样，文件对象由包含访问控制表（ACL）的安全描述体保护。I/O管理器查看安全子系统来决定文件的ACL是否允许进程去访问它的线程正在请求的文件。如果允许，对象管理器将准予访问，并把它返回的文件句柄和给予的访问权限联系起来。如果这个线程或在进程中的另一个线程需要去执行另外的操作，而不是最初请求指定的操作，那么线程就必须打开另一个句柄，它将提示做另外的安全检查。

因为文件对象是表示一个基于内存的共享资源，而表示不是资源本身，所以它有别于其他的

执行体对象。一个文件对象包括的唯一数据是对象句柄，但是文件本身包括将被共享的数据或文本。每次当一个线程打开一个文件句柄时，就创建了一个新的文件，其属性由一组新的句柄指定。例如，属性字节偏移量指的是在文件中下一次将要使用那个句柄做读取或写入操作的位置。每一个为文件打开句柄的线程都有专用的字节偏移量，即使在底层的文件是被共享的。除了当一个进程复制一个文件句柄给另一个进程，或当一个子进程从它的父进程那里继承一个文件句柄以外，文件对象对于进程来说是唯一的。而在这些情况下，两个进程分开引用同一个文件对象的句柄。

尽管一个文件句柄对一个进程可能是唯一的，但在底层的物理资源却不是这样。因此，当使用任何共享的资源时，线程必须保证它对共享文件、文件目录或设备访问的同步。例如，如果一个线程正在写入一个文件，当它打开文件句柄去防止其他的线程在同一时间对该文件写入时，它应该指定独占的写访问。还可以使用Win32的LockFile函数，它可以在写的同时锁定文件的某些部分。

6.4.2 驱动程序对象和设备对象

当线程为文件对象打开一个句柄时，I/O管理器必须根据文件对象名称来决定它将调用哪个（或哪些）驱动程序来处理请求。而且，I/O管理器必须在线程下一次使用同一个文件句柄时可以定位这个信息。下面的系统对象满足这些要求：

- 驱动程序对象代表系统中一个独立的驱动程序，I/O管理器从这些驱动程序对象中获得并且为I/O记录每个驱动程序的调度例程的地址（入口点）。
- 设备对象在系统中代表一个物理的、逻辑的或虚拟的设备并描述了它的特征，例如它所需要的缓冲区的对齐方式和它用来保存即将到来的I/O请求包的设备队列的位置。

当驱动程序被加载到系统中时，I/O管理器将创建一个驱动程序对象，然后它调用驱动程序的初始化例程，该例程把驱动程序的入口点填放到该驱动程序对象中。初始化例程还创建用于每个设备的设备对象，这样使设备对象脱离了驱动程序对象。

当打开一个文件时，文件名包括文件驻留的设备对象的名称。例如，名称\Device\Floppy0\myfile.dat引用软盘驱动器A上的文件myfile.dat。子字符串\Device\Floppy0是Windows 2000/XP内部设备对象的名称，代表那个软盘驱动器。当打开myfile.dat文件时，I/O管理器就创建一个文件对象，并在文件对象中存储一个Floppy0设备的指针，然后给调用者返回一个文件句柄。此后，当调用者使用文件句柄时，I/O管理器能够直接找到Floppy0设备对象。请记住，在Win32应用程序中不能使用Windows 2000/XP内部设备名称。相反，设备名称必须出现在对象管理器的名字空间中的一个特定的目录中（原先的名称是\DosDevices）。这个目录包括到实际的Windows 2000/XP内部设备名称的符号链接。在这个目录中，设备驱动程序负责创建链接以使它们的设备能在Win32应用程序中被访问。通过使用Win32的QueryDosDevice和DefineDosDevice函数，可以检查、甚至可以用编程的方式来改变这些链接。

如图6-9所示，设备对象反过来指向它自己的驱动程序对象，这样I/O管理器就知道在接收一个I/O请求时应该调用哪个驱动程序。它使用设备对象找到代表服务于该设备驱动程序的驱动程

序对象，然后利用在初始请求中提供的功能码来索引驱动程序对象。每个功能码都对应于一个驱动程序的入口点。

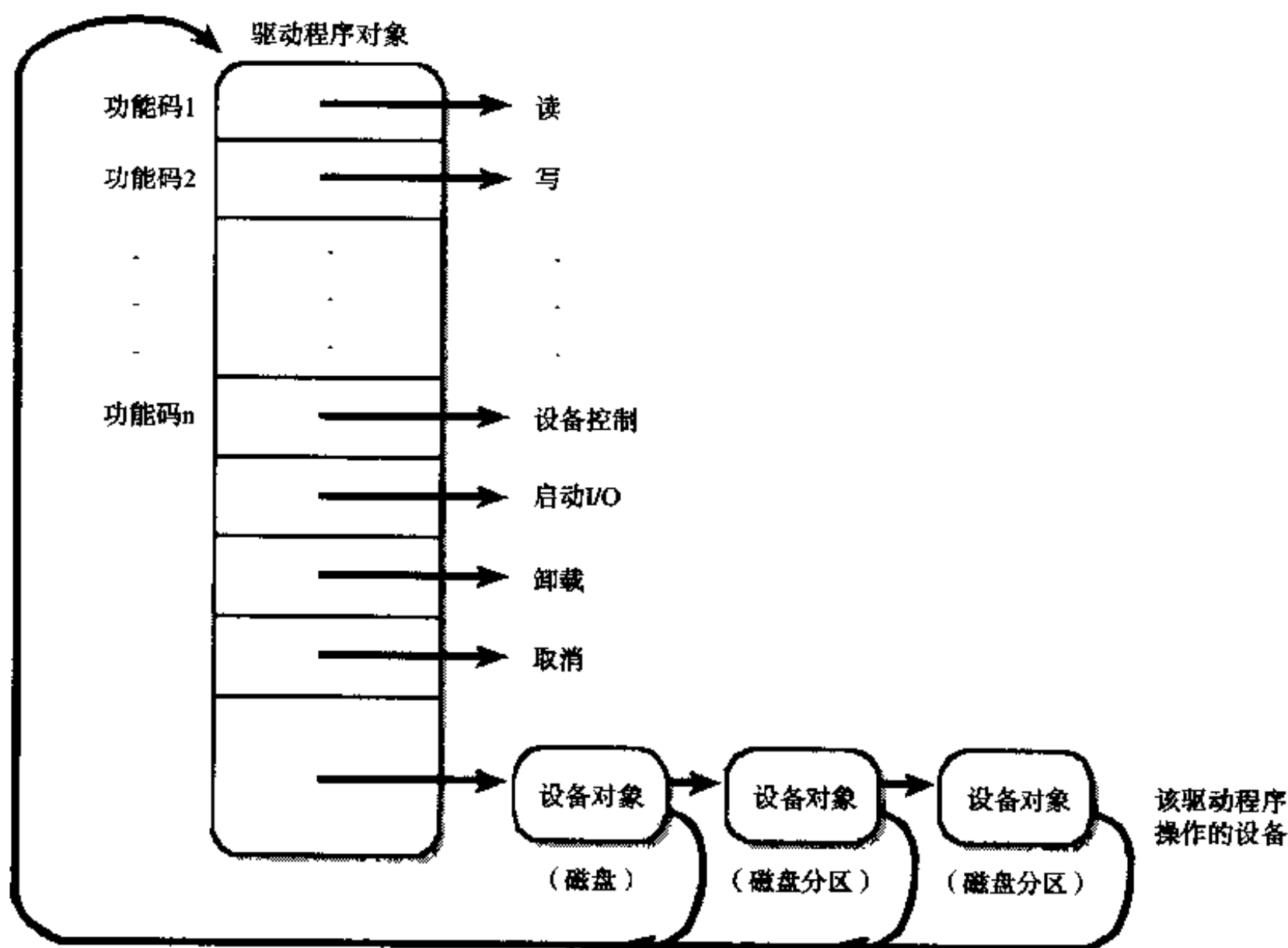


图6-9 驱动程序对象

驱动程序对象通常有多个与它相关的设备对象。设备对象列表代表驱动程序可以控制的物理设备、逻辑设备和虚拟设备。例如，硬盘的每个分区都有一个独立的包含具体分区信息的设备对象。然而，相同的硬盘驱动程序被用于访问所有的分区。当一个驱动程序从系统中被卸载时，I/O管理器就会使用设备对象队列来确定哪个设备由于取走了驱动程序而受到了影响。

6.4.3 I/O请求包

IRP是I/O系统用来存储处理I/O请求所需信息的地方。当线程调用I/O服务时，I/O管理器就构造一个IRP来表示在整个系统I/O进展中要进行的操作。I/O管理器在IRP中保存一个指向调用者文件对象的指针。

图6-10说明了IRP与前面几节描述的文件、设备和驱动程序对象之间的关系。尽管这个例子显示了对一个单层设备驱动程序的I/O请求，但大多数I/O操作都不是这样直接进行的，往往要涉及一个或多个分层的驱动程序。这种情况将在本节之后说明。

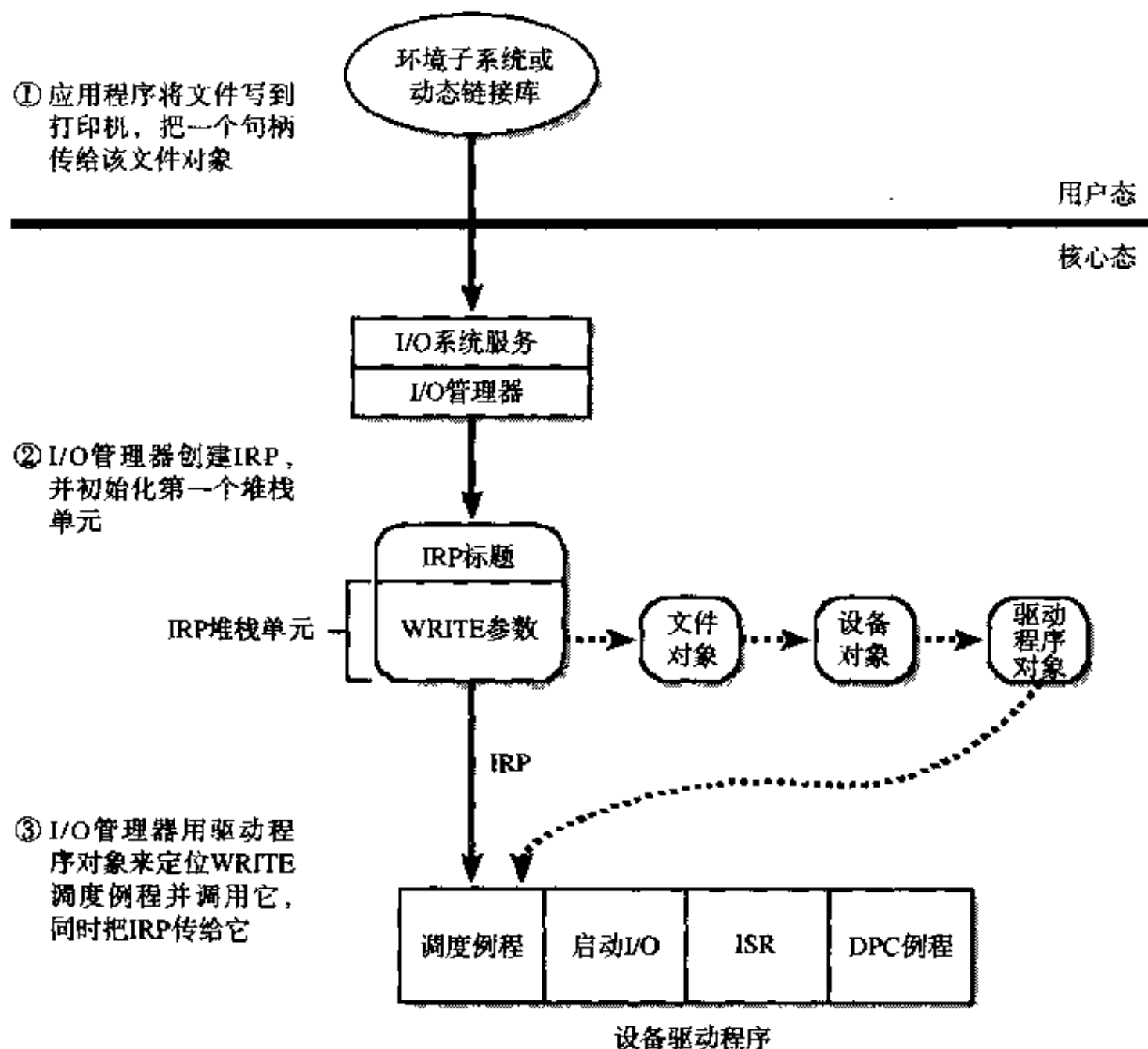


图6-10 单层驱动程序中一个I/O请求涉及的数据结构

IRP由两部分组成：固定部分（称作标题）和一个或多个堆栈单元。固定部分信息包括：请求的类型和大小、是同步请求还是异步请求、用于缓冲I/O的指向缓冲区的指针和随着请求的进展而变化的状态信息。IRP的堆栈单元包括一个功能码、功能特定的参数和一个指向调用者文件对象的指针。

在处于活动状态时，每个IRP都存储在与请求I/O的线程相关的IRP队列中。如果一个线程终止或者被终止时还拥有未完成的I/O请求，这种安排就允许I/O系统找到并释放未完成的IRP。

6.5 Windows 2000/XP的设备驱动程序

Windows 2000/XP支持多种类型的设备驱动程序和编程环境，在同一种驱动程序中也存在不同的编程环境，具体取决于硬件设备。本章主要讨论核心模式的驱动程序，核心驱动程序的种类很多，主要分为以下几种：

- 文件系统驱动接受访问文件的I/O请求，主要是针对大容量设备和网络设备。
- 同Windows 2000/XP的PnP管理器和电源管理器有关的设备驱动程序，包括大容量存储设备、

协议栈和网络适配器等。

- 为Windows NT编写的设备驱动程序，可以在Windows 2000/XP中工作，但是一般不具备电源管理和PnP的支持，会影响整个系统的电源管理和PnP管理的能力。
- Win32子系统显示驱动程序和打印驱动程序将把与设备无关的图形（GDI）请求转换为设备专用请求。这些驱动程序的集合被称为“核心态图形驱动程序”。显示驱动程序的集合被称为“核心态图形驱动程序”。显示驱动程序与视频小端口（miniport）驱动程序是成对的，用来完成视频显示支持。每个视频小端口驱动程序为与它关联的显示驱动程序提供硬件级的支持。
- 符合Windows驱动程序模型的WDM驱动程序，包括对PnP、电源管理和WMI的支持。WDM在Windows 2000/XP、Windows 98和Windows ME中都是被支持的，因此在这些操作系统中是源代码级兼容的，在许多情况下是二进制兼容的。有三种类型的WDM驱动程序：
 - 总线驱动程序（bus driver）管理逻辑的或物理的总线，例如PCMCIA、PCI、USB、IEEE 1394和ISA，总线驱动程序需要检测并向PnP管理器通知总线上的设备，并且能够管理电源。
 - 功能驱动程序（function driver）管理具体的一种设备，对硬件设备进行的操作都是通过功能驱动程序进行的。
 - 过滤器驱动程序（filter driver）与功能驱动程序协同工作，用于增加或改变功能驱动程序的行为。

除了以上这些驱动程序类型外，Windows 2000/XP还支持一些用户模式的驱动程序：

- 虚拟设备驱动程序（VDD）通常用于模拟16位MS-DOS应用程序。它们捕获MS-DOS应用程序对I/O端口的引用，并将其转化为本机Win32 I/O函数。因为Windows 2000/XP是一个完全受保护的操作系统，用户态MS-DOS应用程序不能直接访问硬件，而必须通过一个真正的核心设备驱动程序。
- Win32子系统的打印驱动程序将与设备无关的图形请求转换为打印机相关的命令，这些命令再发给核心模式的驱动程序，例如并口驱动（Parport.sys）、USB打印机驱动（Usbprint.sys）等。

除了总线驱动、功能驱动、过滤器驱动外，硬件支持驱动可以分为以下类型：

- 类驱动程序（class driver）为某一类设备执行I/O处理，例如磁盘、磁带或光盘。
- 端口驱动程序（port driver）实现了对特定于某一种类型的I/O端口的I/O请求的处理，例如SCSI。
- 小端口驱动程序把对端口类型的一般的I/O请求映射到适配器类型。例如，一个特定的SCSI适配器。

图6-11给出了一个例子，用以说明这些设备驱动程序是如何工作的。文件系统驱动程序收到向特定文件写数据的请求，它将此请求转换为向磁盘指定的“逻辑”位置写字节的请求，然后再把这个请求传递给一个简单的磁盘驱动程序。这个磁盘驱动程序再依次把请求转换为磁盘上的柱

面/磁道/扇区，并且操作磁头来重现数据。

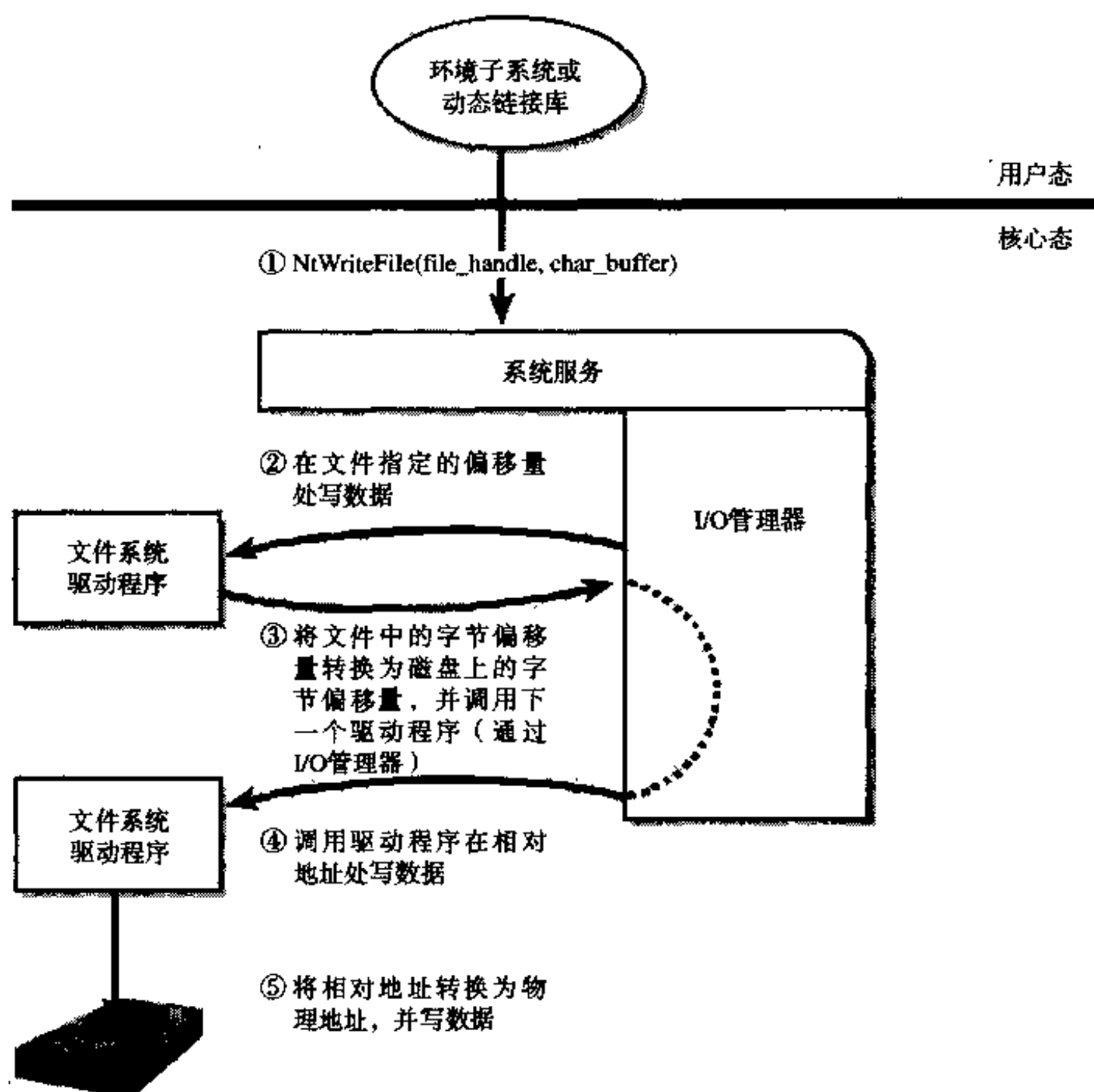


图6-11 文件系统驱动和磁盘驱动的层次

图6-11说明了在两层驱动程序之间的工作分界线。I/O管理器接受了与一个特殊文件的开始部分有关的写请求，并将这个请求传递到文件系统驱动程序，这个驱动程序再把写操作从与文件有关的操作转换为开始位置（磁盘上一个扇区的边界）和要读取的字节数。文件系统驱动程序调用I/O管理器把请求传递到磁盘驱动程序，这个驱动程序将请求转换为物理磁盘位置，并且传递数据。

因为所有的驱动程序（包括设备驱动程序和文件系统驱动程序）对于操作系统来说都呈现相同的结构，一个驱动程序可以不经转换当前的驱动程序或I/O系统，就能容易地被插入到分层结构中。例如，通过添加驱动程序，可以使几个磁盘看起来很像非常大的单一的磁盘。在Windows 2000/XP中实际上就存在这样一个驱动程序来提供容错磁盘支持。这个逻辑的、多卷的驱动程序位于文件系统和磁盘驱动程序之间，如图6-12所示。

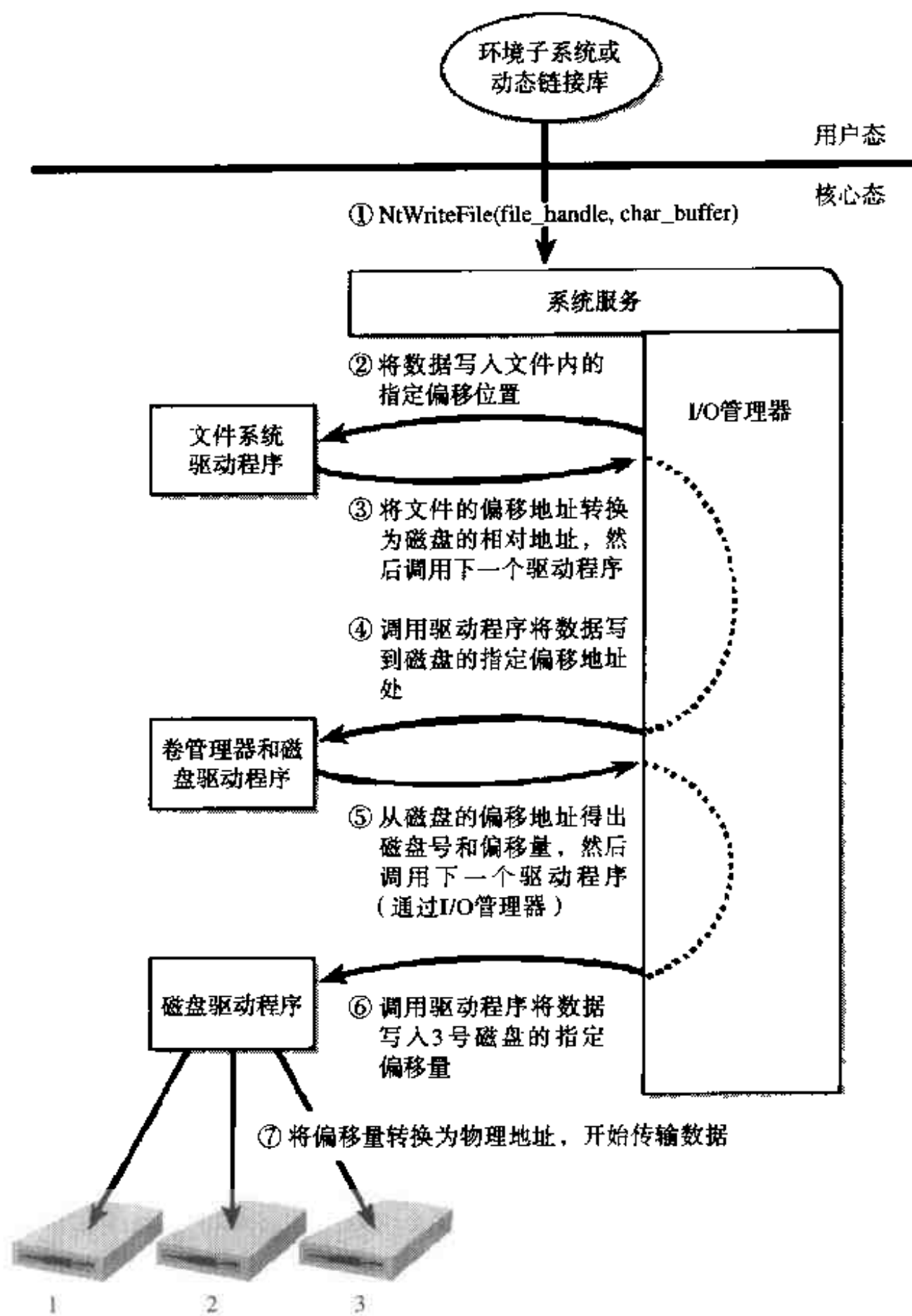


图6-12 添加一个分层驱动程序

6.5.1 驱动程序结构

设备驱动程序包括一组调用处理I/O请求不同阶段的例程。如图6-13所示，主要有五个设备驱动程序例程：

- **初始化例程** 当I/O管理器把驱动程序加载到操作系统中时，它执行驱动程序的初始化例程。

这个例程将创建系统对象。I/O管理器利用这些系统对象去识别和访问驱动程序。

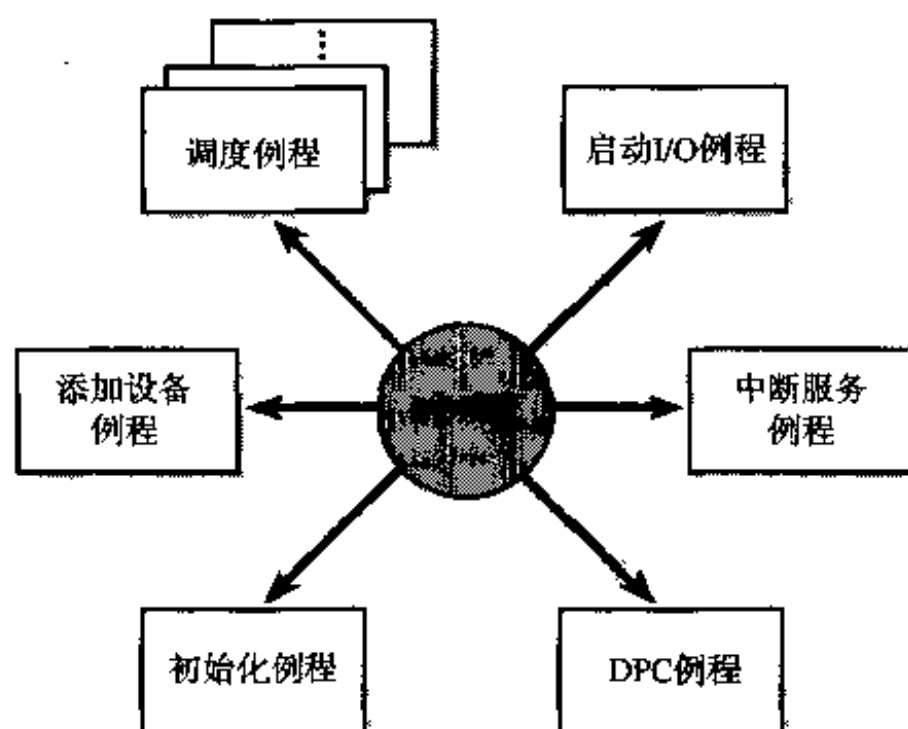


图6-13 主要的设备驱动例程

- **添加设备例程** 用于支持PnP管理器的操作。
- **一系列调度例程** 调度例程是设备驱动程序提供的主要函数。例如打开、关闭、读取、写入以及设备、文件系统或网络支持的任何其他功能。当被调用去执行一个I/O操作时，I/O管理器产生一个IRP，并且通过某个驱动程序的调度例程调用驱动程序。
- **启动I/O例程** 驱动程序可以使用启动I/O例程来初始化与设备之间的数据传输。
- **中断服务例程（ISR）** 当一个设备中断时，内核的中断调度程序把控制转交给这个例程。在Windows 2000/XP的I/O模型中，ISR运行在高级的设备中断请求级（IRQL）上，所以它们越简单越好，以避免对低优先级中断产生不希望的阻塞。ISR将运行在低IRQL的延迟过程调用（DPC）队列上，以执行中断处理的剩余部分（只有用于中断驱动设备的驱动程序才有ISR，例如文件系统就没有ISR）。
- **中断服务DPC例程** DPC例程执行在ISR执行以后的大部分设备中断处理工作。DPC例程在低于ISR的IRQL的时候执行，从而避免对其他中断产生不希望的阻塞。DPC例程初始化I/O完成并启动关于设备的下一个队列的I/O操作。

尽管下面的例程没有列出，但是可以在很多类型的设备驱动程序中找到。

- **一个或多个完成例程** 分层驱动程序可能会有完成例程，通过它一个较低层的驱动程序确定何时完成对一个IRP的处理。例如，当设备驱动程序完成了与文件的数据传输以后，I/O管理器将调用文件系统的完成例程。该完成例程通知文件系统关于操作的成功、失败或取消，并且允许文件系统执行清理操作。
- **取消I/O例程** 如果某个I/O操作可以被取消，驱动程序就可以定义一个或多个取消I/O例程。I/O管理器调用什么样的取消例程，取决于I/O操作在被取消时已进行到什么程度。
- **卸载例程** 卸载例程释放任何驱动程序正在使用的系统资源，以使I/O管理器能从内存中删

除它们。当系统运行时，驱动程序可以被加载或卸载。

- **系统关闭通知例程** 这个例程允许驱动程序在系统关闭时做清理工作。
- **错误记录例程** 当意外错误发生时（例如，当磁盘分区被损坏时），驱动程序的错误记录例程将记录发生的事情，并通知I/O管理器。I/O管理器把这个信息写入错误记录文件。

6.5.2 同步

驱动程序必须同步执行它们对全局驱动程序数据的访问，这有两个主要原因：

- 驱动程序的执行可以被高优先级的线程抢先，或时间片（或时间段）到时被中断，或被其他中断所中断。
- 在多处理器系统中，Windows 2000/XP能够同时在多个处理器上运行驱动程序代码。

若不能同步执行，就会导致相应错误的发生。例如，因为设备驱动程序代码运行在低优先级的IRQL上，所以当调用者初始化一个I/O操作时，可能被设备中断请求所中断，从而导致在它的设备驱动程序正在运行时让设备驱动程序的ISR去执行。如果设备驱动程序正在修改其ISR也要修改的数据，例如设备寄存器、堆存储器或静态数据，则在ISR执行时，数据可能被破坏。

要避免这种情况发生，为Windows 2000/XP编写的设备驱动程序就必须对它和它的ISR对共享数据的访问进行同步控制。在尝试更新共享数据之前，设备驱动程序必须锁定所有其他的线程（或CPU，在多处理器系统的情况下），以防止它们修改同一个数据结构。

当设备驱动程序访问其ISR也要访问的数据时，Windows 2000/XP的内核提供了设备驱动程序必须调用的特殊的同步例程。当共享数据被访问时，这些内核同步例程将禁止ISR的执行。在单CPU系统中，在更新一个结构之前，这些例程将IRQL提高到一个指定的级别。然而，在多处理器系统中，因为一个驱动程序能同时在两个或两个以上的处理器上执行，以这种技术就不足以阻止其他的访问。因此，另一种被称为“自旋锁”的机制被用来锁定来自指定CPU的独占访问的结构（“自旋锁”在第3章的“内核同步”一节中有详细描述）。

到目前为止，应该意识到尽管ISR需要特别的关注，但一个设备驱动程序使用的任何数据将面临运行于另一个处理器上的相同的设备驱动程序的访问。因此，用设备驱动程序代码来同步它对所有全局的或共享数据（或任何到物理设备本身的访问）的使用是很危险的。如果数据被ISR使用，设备驱动程序就必须使用内核同步例程或者使用一个内核锁。

6.6 Windows 2000/XP的I/O处理

在了解了驱动程序的结构和类型以及支持该结构和类型的数据结构之后，现在来看I/O请求是如何在系统中传递的。一个I/O请求会经过若干个处理阶段，而且根据请求是指向由单层驱动程序操作的设备还是一个经过多层驱动程序才能到达的设备，它经过的阶段也有所不同。因为处理的不同进一步依赖于调用者是否指定了同步I/O还是异步I/O，所以先了解一下这同种I/O类型的处理以及其他几种不同类型的I/O。

6.6.1 I/O的类型

应用程序在发出I/O请求时可以设置不同的选项，例如设置同步I/O或者异步I/O，设置应用程序获取I/O数据的方式等。

1. 同步I/O和异步I/O

应用程序发出的大多数I/O操作都是“同步”的，也就是说，设备执行数据传输并在I/O完成时返回一个状态码，然后程序就可以立即访问被传输的数据。ReadFile和WriteFile函数使用最简单的形式调用时是同步执行的，在把控制返回给调用程序之前，它们完成一个I/O操作。

“异步I/O”允许应用程序发布I/O请求，然后当设备传输数据的同时，应用程序继续执行。这类I/O能够提高应用程序的吞吐率，因为它允许在I/O操作进行期间，应用程序继续其他的工作。要使用异步I/O，必须在Win32的CreateFile函数中指定FILE_FLAG_OVERLAPPED标志。当然，在发出异步I/O操作请求之后，线程必须小心地不访问任何来自I/O操作的数据，直到设备驱动程序完成数据传输。线程必须通过等待一些同步对象（无论是事件对象、I/O完成端口或文件对象本身）的句柄，使它的执行与I/O请求的完成同步。当I/O完成时，这些同步对象将会变成有信号状态（关于如何使用这些对象的详细信息，请参阅Platform SDK文档）。

与I/O请求的类型无关，由IRP代表的内部I/O操作都将被异步执行；也就是说，一旦一个I/O请求已经被启动，设备驱动程序就返回I/O系统。I/O系统是否返回调用程序取决于文件是否为异步I/O打开的。

可以使用Win32的HasOverlappedIoCompleted函数去测试挂起的异步I/O的状态。如果正在使用I/O完成端口，则可用GetQueuedCompletionStatus函数。

2. 快速I/O

快速I/O是一个特殊的机制，它允许I/O系统不产生IRP而直接到文件系统驱动程序或高速缓存管理器去执行I/O请求（快速I/O将在随后的章节中详细描述）。

3. 映射文件I/O和文件高速缓存

映射文件I/O是I/O系统的一个重要特性，是I/O系统和内存管理器共同产生的（关于如何实现映射文件的详细信息，请参阅相关章节）。“映射文件I/O”是指把磁盘中的文件视为进程的虚拟内存的一部分。程序可以把文件作为一个大的数组来访问，而无需做缓冲数据或执行磁盘I/O的工作。程序访问内存，同时内存管理器利用它的页面调度机制从磁盘文件中加载正确的页面。如果应用程序向它的虚拟地址空间写入数据，内存管理器就把更改作为正常页面调度的一部分写回到文件中。

通过使用Win32的CreateFileMapping和Map ViewOfFile函数，映射文件I/O对于用户态是可用的。在操作系统中，映射文件I/O被用于重要的操作中，例如文件高速缓存和映像活动（加载并运行可执行程序）。其他重要的使用映射文件I/O的程序还有高速缓存管理器。文件系统使用高速缓存管理在虚拟内存中的映像文件数据，从而为I/O绑定程序提供了更快的响应时间。当调用者使用文件时，内存管理器将把被访问的页面调入内存。尽管多数高速缓存系统在内存中分配固定数量的字节给高速缓存文件，但Windows 2000/XP高速缓存的增大或缩小取决于可以获得的内存

有多少。这种大小的变化是可能的，因为高速缓存管理器依赖于内存管理器来自动地扩充（或缩小）高速缓存的数量，它使用正常工作集机制来实现这一功能。通过利用内存管理器的页面调度系统，高速缓存避免了重复内存管理器已经执行了的工作。

4. 分散/集中I/O

Windows 2000/XP同样支持一种特殊类型的高性能I/O，它被称作“分散/集中”（scatter/gather），可通过Win32的ReadFileScatter和WriteFileScatter函数来实现。这些函数允许应用程序执行一个读取或写入操作，从虚拟内存的多个缓冲区读取数据并写到磁盘上文件的一个连续区域里。要使用分散/集中I/O，文件必须以非高速缓存I/O方式打开，被使用的用户缓冲区必须是页对齐的，并且I/O必须被异步执行（重叠）。

6.6.2 对单层驱动程序的I/O请求

这一节将讨论对单层核心态设备驱动程序的同步I/O请求。处理对单层驱动程序的同步I/O包括以下六步：

- 1) I/O请求经过子系统DLL。
- 2) 子系统DLL调用I/O管理器的NtWriteFile服务。
- 3) I/O管理器以IRP的形式给驱动程序（这里指设备驱动程序）发送请求。
- 4) 驱动程序启动I/O操作。
- 5) 在设备完成了操作并且中断CPU时，设备驱动程序服务于中断。
- 6) I/O管理器完成I/O请求。

这六步如图6-14所示。

下面进一步看看中断处理和I/O完成。

1. 处理一个中断

在I/O设备完成数据传输之后，它将产生中断并请求服务，这样Windows 2000/XP的内核、I/O管理器和设备驱动程序都将被调用。

当I/O设备中断发生时，处理器将控制转交给内核陷阱处理程序，内核陷阱处理程序将在它的中断向量表中搜索定位用于设备的ISR。Windows 2000/XP上的ISR用两个步骤来典型地处理设备中断。当ISR被首次调用时，它通常只在设备IRQL上停留获得设备状态所必需的一段时间，最后停止设备的中断。然后它使一个DPC排入队列，并退出服务例程，清除中断。过一段时间，在DPC例程被调用时，设备完成对中断的处理。完成之后，设备将调用I/O管理器来完成I/O并处理IRP。它也可以启动下一个正在设备队列中等待的I/O请求。

使用DPC来执行大多数设备服务的优点是，任何优先级位于设备IRQL和Dispatch/DPC IRQL之间被阻塞的中断允许在低优先级的DPC处理发生之前发生。因而中间优先级的中断就可以更快地得到服务。

2. 完成I/O请求

当设备驱动程序的DPC例程执行完以后，在I/O请求可以考虑结束之前还有一些工作要做。I/O处理的第三阶段称作“I/O完成”（I/O completion），它因I/O操作的不同而不同。例如，全部

的I/O服务都把操作的结果记录在由调用者提供的数据结构“I/O状态块”(I/O status block)中。与此相似,一些执行缓冲I/O的服务要求I/O系统返回数据给调用线程。

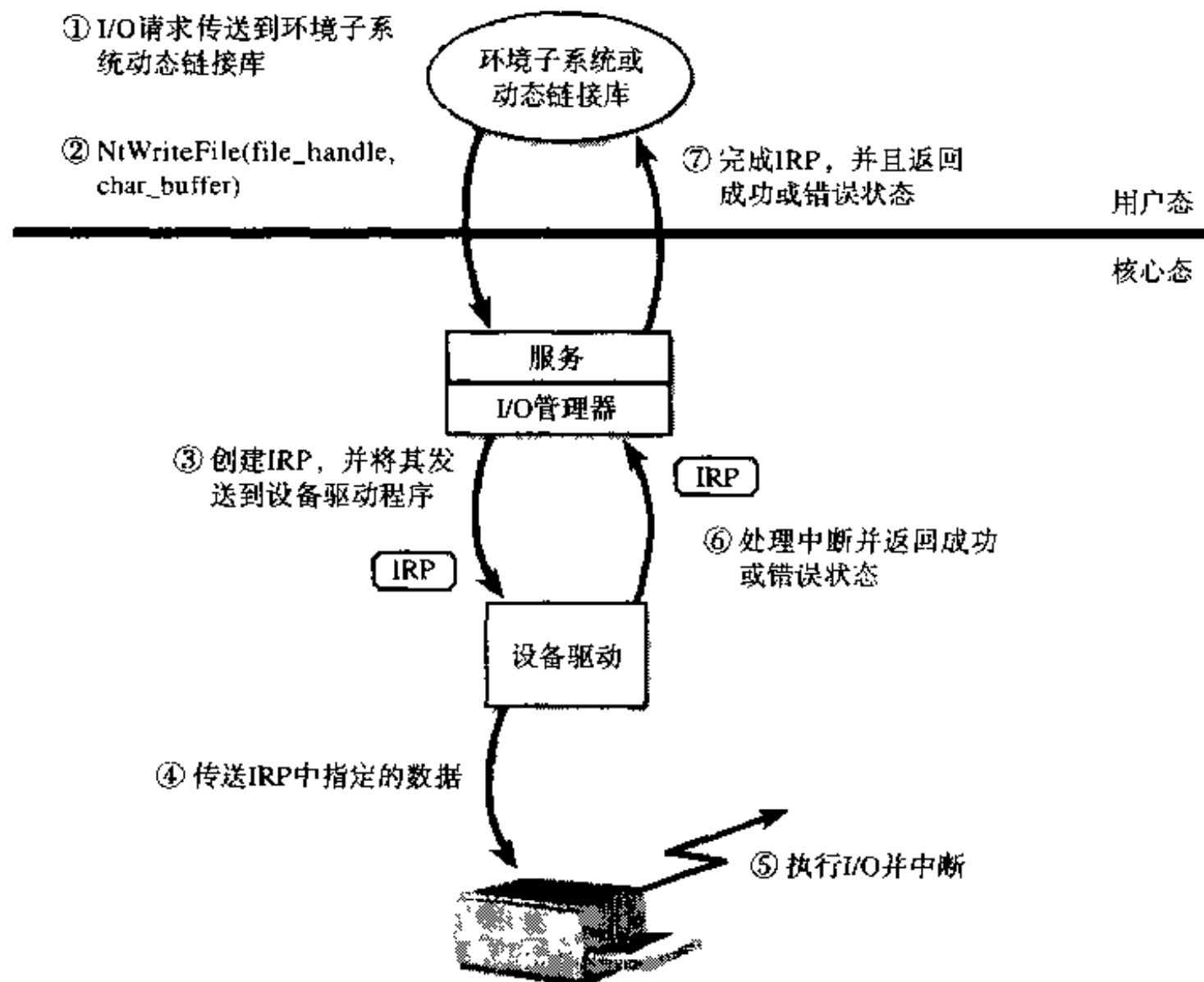


图6-14 一个同步请求的处理

在上述两种情况中, I/O系统必须把一些存储在系统内存中的数据复制到调用者的虚拟地址空间中。要获得调用者的虚拟地址, I/O管理器必须在调用者线程的上下文中进行数据传输,而此时调用者进程是当前处理器上活动的进程,调用者线程正在处理器上执行。I/O管理器通过在线程中执行一个核心态的异步过程调用(APC)来完成这个操作。

APC在特定线程的描述表中执行,而DPC在任意线程的描述表中执行,这就意味着DPC例程不能涉及用户态进程的地址空间。要注意的是, DPC具有比APC更高的软件中断优先级。

接下来当线程开始在较低的IRQL上执行时,挂起的APC被提交运行。内核把控制权转交给I/O管理器的APC例程,它将把数据(如果有)和返回的状态复制到最初调用者的地址空间,释放代表I/O操作的IRP,并将调用者的文件句柄(或调用者提供的事件或I/O完成端口)设置为有信号状态。现在才可以考虑完成I/O。在文件(或其他对象)句柄上等待的最初调用者或其他线程都将从它们的等待状态中被唤醒并准备再一次执行。

关于I/O完成最后要注意的是:异步I/O函数ReadFileEx允许调用者提供用户态APC作为参数。如果调用者这样做了, I/O管理器将在I/O完成的最后一步为调用者清除这个APC,这个特性允许

调用者在I/O请求完成时指定一个将被调用的子程序。正如在Platform SDK文档中对这些函数解释的那样，用户态APC完成例程在请求线程的描述表中执行，并且只有当线程进入可报警等待状态时才可以被传送（例如调用Win32 `sleepEx/WaitForSingleObjectEx`或`WaitForMultipleObjectsEx`函数）。

6.7 小结

I/O系统定义了Windows 2000/XP上的I/O处理模型，并且执行公用的或被多个驱动程序请求的功能。它主要负责创建代表I/O请求的IRP和引导通过不同驱动程序的包，在完成I/O时向调用者返回结果。I/O管理器通过使用I/O系统对象来定位不同的驱动程序和设备，这些对象包括驱动程序对象和设备对象。内部的Windows 2000/XP I/O系统以异步操作方式获得高性能，并且向用户态应用程序提供同步和异步I/O功能。

设备驱动程序不仅包括传统的硬件设备驱动程序，还包括文件系统、网络 and 分层过滤器驱动程序。通过使用公用机制，所有驱动程序都具有相同的结构，并以相同的机制在彼此之间和I/O管理器通信。I/O系统接口允许使用高级语言写驱动程序，以节省开发时间并增强它们的可移植性。因为驱动程序在操作系统中以相同的结构出现，所以它们可以被分层，即把一层放在另一层上来达到模块化，并可以减少在驱动程序之间的复制。同样，所有的Windows 2000/XP设备驱动程序都应被设计成能够在多处理器系统下工作。

PnP管理器可以动态地检测硬件设备安装和卸载，并且构建内部的设备树来控制设备驱动程序的安装。电源管理器可以把整个系统或单个硬件设备转入到低能耗状态，以节省能源消耗。

习题

- 6.1 I/O系统在整个操作系统中所起的作用和地位是什么？
- 6.2 什么叫做设备无关性？
- 6.3 在I/O系统中引入缓冲的主要原因是什么？
- 6.4 设备驱动程序应该具有哪些特点和功能？
- 6.5 为什么要引入设备独立性？如何实现设备独立性？
- 6.6 说明SPOOL系统的特点以及其工作方式。
- 6.7 I/O系统的层次结构和每层的功能是什么？
- 6.8 衡量I/O系统性能的标准有哪些？
- 6.9 说明Windows2000/XP的I/O系统的设计目标和结构特点。
- 6.10 请给出在Windows2000/XP中，一个典型的I/O请求的流程。
- 6.11 在Windows2000/XP中，I/O管理器的作用是什么？
- 6.12 什么是I/O请求包（IRP），在Windows2000/XP的I/O系统中起什么作用？
- 6.13 Windows2000/XP是如何实现对PnP的支持的？
- 6.14 简要说明Windows2000/XP对高级电源管理的支持。
- 6.15 说明在Windows2000/XP中同步I/O和异步I/O的区别。

第 7 章

网 络

第 ⑦ 章

网 络

从20世纪70年代以来，计算机网络在整个计算机产业中掀起了一场革命。随着网络的出现，计算机不再只是单机运行，而是通过网络进行通信协作。计算机网络是一个复杂的系统，存在大量的技术，许多组织已经独立地制订了网络标准，但彼此并不完全兼容。由于有多种技术被用来连接不同的计算机网络，因此可能有多种可能的连接方式，导致网络变得十分复杂。操作系统负责管理复杂的计算机系统，网络自然成为不可缺少的一部分。早期的计算机都是单机运行，因此不存在网络的概念，但是网络的飞速发展使得操作系统不得不引入复杂而庞大的网络模块，可以说网络子系统是操作系统中变化最大的部分。

微软公司在设计Windows 2000时，就把网络作为一种基本服务设施集成在操作系统中，而且主要的网络支持已经与I/O系统和Win32 API集成于一体。网络软件有四种基本类型，分别是服务、API、协议和网络适配器的设备驱动程序，它们各为一层，由上至下构成了网络栈。Windows 2000为每一层提供了良好定义的接口，除了使用由Windows 2000提供的各种不同的API、协议和设备驱动程序，用户还可以自己开发，以用于扩展操作系统的网络能力。

本章，我们将从网络的基本原理开始，介绍OSI参考模型和TCP/IP参考模型。然后，我们简要描述Windows 2000提供的网络API并说明它们是怎样实现的。你会明白网络资源名字解析怎样工作，以及协议驱动程序（Protocol Driver）是怎样实现的。最后，我们简述一下Windows 2000的层次化网络服务，如活动目录这样的目录服务和文件复制服务（File Replication Service，FRS）。

7.1 网络基本原理

网络软件的职责是从一个系统的应用程序中取得一个请求，并将其传送给另一系统，这个请求在远程系统上被执行并将结果返回给最初的系统。由于网络软件一般是按分层思想设计的，所以在整个过程中，请求的传输格式会被转换多次。一个高级请求，就像“从z机器上的文件y中读取x字节”，它不需要给出具体实现的底层细节，相应的服务知道如何能够到达z机器，以及对方使用的是何种通信软件。于是跨网络请求会因此从上层到下层，再从下层到上层转换多次。例如，一个请求被切割成好几个短分组信息。当请求到达另一端时，会接受完整性校验检查，再解码，然后再传给相应的组件执行。最后，执行结果经过编码，然后从网络上送回发送请求的机器。

为了便于不同的计算机制造商对他们的网络软件进行标准化和集成，1974年国际标准化组织（ISO）定义了一个系统之间信息传输的软件模型：开放系统互连参考模型（OSI）。OSI模型定义

了七层软件，见图7-1。

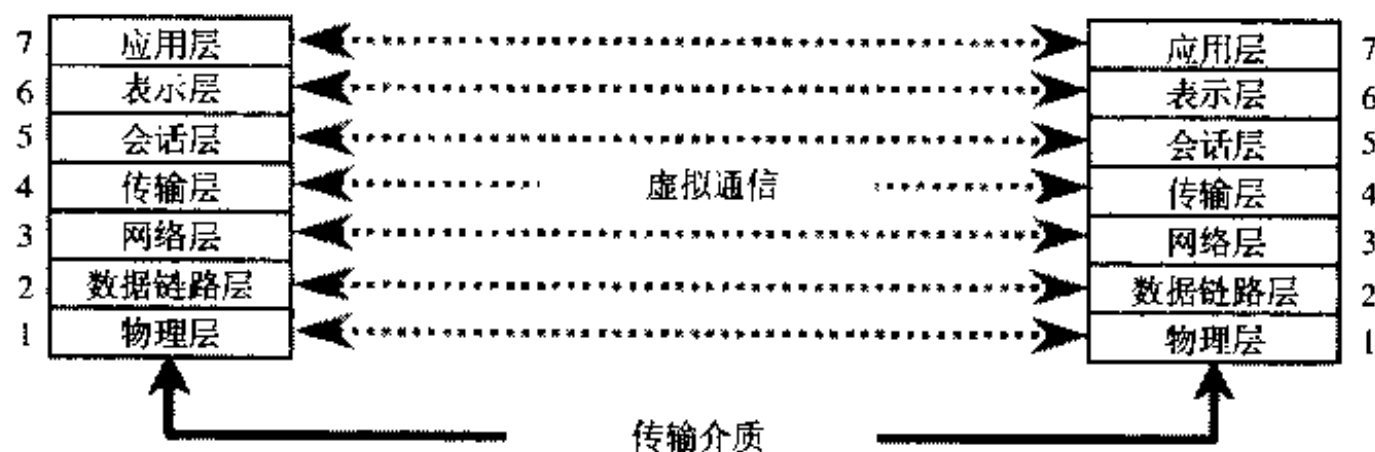


图7-1 OSI参考模型

OSI参考模型是一种理想化的方案，只有少数系统完全实现了这一模型，然而，OSI参考模型为示范网络的功能结构提供了可行的机制。两台机器在相同的层次上，使用同一种语言（协议）对话。然而，网络传输实际上是通过客户机的每一层往下传，然后在目标机上逐层上传，直到相应的一层能够理解并应答客户机的请求。

7.1.1 OSI参考模型

在OSI模型中，每一层的功能是向上层提供服务，而对下层的实现进行抽象。下面简要地说明OSI模型每一层的功能：

- **应用层** 处理两个网络应用程序之间的信息传输，包括安全校验、对方机器的身份识别以及数据交换的初始化等功能。
- **表示层** 负责所传输消息的语法和语义的分析，处理数据的格式化，包括文本行是否携带回车/换行（CR/LF）还是仅有回车（CR），数据是否经过压缩还是编码等。
- **会话层** 管理相互协作的应用程序之间的连接，包括高层同步以及对正在发送信息的应用程序和正在接受信息的应用程序的监控。
- **传输层** 从会话层接受数据，传递给网络层，并确保到达对方的信息正确无误。在客户端，传输层将信息封装在已编号的分组内，这样接收方能够按次序重组收到的数据。在接收端，传输层将收到的乱序的数据包重新排序，传递给会话层。传输层向会话层屏蔽了硬件之间的差别所带来的影响。
- **网络层** 负责建立分组头，处理路由、拥塞控制以及网络互连。网络层是整个OSI模型中能够知道网络拓扑结构的最高层，所谓网络拓扑是指在网络中机器的物理配置，还包括网络带宽限制等。
- **数据链路层（DLL）** 它有发送和接收两项主要任务。在发送方，DLL需负责将指令、数据等包装到帧中。帧是DLL层生成的结构，它包含足够的信息，确保数据可以安全地通过本地局域网到达目的地。DLL的另一个职责是重新组织从物理层收到的数据比特流。DLL还要提供数据有效传输的端到端连接。
- **物理层** 这一层负责传送比特流。它从数据链路层接收数据帧，并将帧的结构和内容串行

发送，即每次发送一个比特，然后这些数据流被传输给DLL重新组合成数据帧。

图7-1中的虚线表示一个请求传输至远端机器时所用到的协议。就像先前陈述的，每一层可以看作只与另一台机器的同一层对话，并且使用公共的协议。一个请求自上而下再自下而上所经过的协议层构成了一个协议的集合，我们称之为协议栈。

7.1.2 TCP/IP参考模型

TCP/IP模型是随因特网而诞生的参考模型，而因特网是由ARPANET不断发展和改进后的产物。ARPANET是由美国国防部赞助的研究网络，它通过租用的电话线将大学和政府部门连在一起。由于卫星和无线网络的出现，原先的协议在互连时出现了无法解决的问题，因此，随后经历了一番研究和实践，新的参考体系结构具有无缝隙地连接多个网络的能力。这一模型称之为TCP/IP参考模型。TCP/IP参考模型与OSI模型的比较见图7-2。

1. 互连网络层

互连网络层的功能等同于OSI模型中的网络层。在该层中，有几种不同的协议用于分组的路由和发送。通常，网络通过路由器达到互连，而路由器就工作在该层，负责转发来自不同网络的分组。以下介绍该层用到的一些协议。

(1) 网际协议（IP）

网际协议是基于面向无连接的分组交换协议，承担寻址和路由选择的任务。当一个分组向下层传输的时候，它被该层的协议加上一个头部，里面保存的信息能够使分组在网络中应用动态路由表进行路由选择。由于该协议是面向无连接的，所以在任何一个分组发送完毕后，并不需要接收方有任何的确认消息。另外因下层的需要，IP协议还负责分组的拆分和重组。每个IP分组都包含一个源地址字段和目的地址、协议标识符、头校验和以及生存期（TTL）等信息。其中，TTL字段告诉路由器该分组在到达目的地之前还能在网络上存在多久，它一般以减法计数器或时钟技术的形式工作。当分组经过路由器的时候，该分组的TTL会减小。例如，一个分组的TTL值为128，这就代表它还能生存128秒或者128跳等。TTL的目的在于终止那些已丢失的或损坏的数据分组在网络上无休止的徘徊。当TTL的值减到0时，分组将从网络上消失。

(2) 地址解析协议（ARP）

在IP分组能够被转发到其他主机之前，必须先知道接收方的硬件地址。ARP协议负责由IP地址到硬件地址的转换（MAC地址）。如果ARP协议在缓存中找不到相应的地址映射，它将通过广播询问该地址的映射。网络上的任何一台主机都会收到此请求，如果它们之中有一个知道该地址的转换，则将转换结果发送给询问请求者。然后，分组就能送往目的地，同时ARP协议将得到的地址映射保存在缓存中。

(3) 反向地址解析协议（RARP）

RARP服务器维护一个机器号的数据库，其中的数据以ARP表的形式组织，它由系统管理员

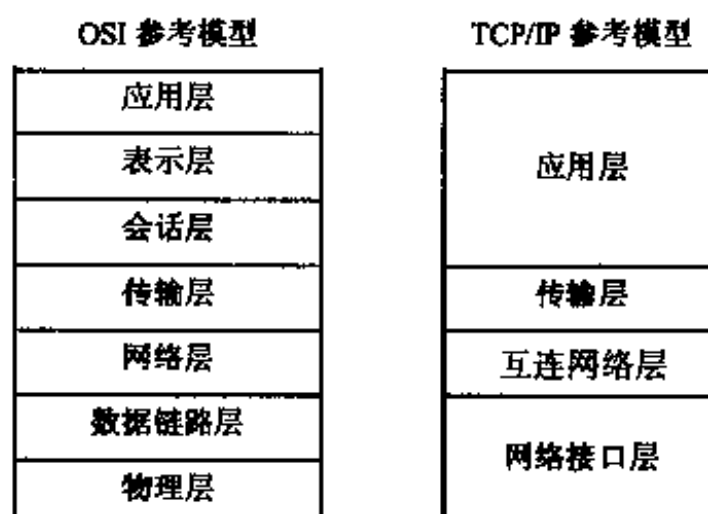


图7-2 OSI参考模型与TCP/IP参考模型

创建。与ARP协议相反，RARP协议完成由硬件地址到IP地址的转换。当RARP服务器接收到一个由硬件地址到IP地址的解析请求时，它将根据该硬件地址搜索数据库，并将结果返回请求节点。

(4) 因特网差错控制协议 (ICMP)

ICMP用于向IP或更上层提供传输状态信息。路由器利用ICMP进行数据流或数据传输速率的控制。如果数据流对某一路由器的传输过快，则它请求发送方减缓数据发送。

ICMP协议用两种方式提供信息：一种是主动报告错误，另一种是轮询。

2. 传输层

它与OSI模型中传输层的位置相同，主要负责建立和维持两个主机之间的端到端通信。传输层提供数据传送的确认、流控制、分组的排序及其重传。根据传输的需求，该层可以选择基于面向连接的传输控制协议 (TCP) 或者基于面向无连接的用户数据报协议 (UDP)。

(1) TCP

TCP负责可靠的面向连接的数据传输。在两点间进行传输之前需要建立一条连接（也称为会话、虚电路）。TCP使用三次握手协议建立连接。这一过程也建立了传输端口号以及两端的初始序列号 (ISN: Initial Sequence Number)。三次握手协议包含以下三个步骤：

- 1) 请求方发送一个带有端口号和ISN的分组给接收方。
- 2) 接收方发回一个带有请求方和接收方的ISN的确认给请求方。
- 3) 请求方发回一个带有接收方的ISN给接收方。

为了维护一个可靠的连接，每个分组必须包含：

- 源和目的地的TCP端口号。
- 拆分了的消息中的序列号。
- 校验和用以保证发送没有错误。
- 确认号用以保证已拆分的消息被接收到。
- TCP滑动窗口。

(2) 端口、套接字和滑动窗口

协议端口号用于在每台机器上（应用层）定位一个指定的应用程序或进程。如同IP地址用于识别网络中的主机一样，端口地址用于识别传输层上的应用程序，这样也就在两台机器上的应用程序之间提供了一个完全连接。应用程序和服务程序可以应用多达65 536个端口号。TCP/IP应用及其服务使用了前1024个端口号。IANA (Internet Assigned Numbers Authority) 已经指定了这些标准或缺省的端口号。任何客户应用程序可以根据需要动态地制定端口号。端口号和节点地址组成了套接字 (socket)。

应用程序和服务使用套接字来建立连接。如果应用程序需要可靠的数据传输，则套接字选用面向连接的服务 (TCP)。反之，则使用面向无连接的服务 (UCP)。

TCP使用滑动窗口协议传输数据。它定义了接收方在发送确认之前发送方至多能传输多少数据。每台计算机拥有一个发送窗口和一个接收窗口，并且利用其缓冲数据达到高效的通信。滑动窗口协议允许发送方以流的形式传输数据，而无需对每个分组进行确认。这样一来，接收方就需要重组收到的分组。发送方跟踪已经发送的数据，如果在一段时间内没有收到确认，则这些分组就需要重传。

(3) UDP

面向无连接的UDP协议负责端到端的数据传输。与TCP相比，UDP不需要建立连接。它试图发送数据并确认目的地真的收到数据。UDP适合发送无需可靠连接的短消息。UDP使用的端口不同于TCP的端口，因此它们可以使用同一个端口号。

3. 应用层

它对应于OSI模型中的会话层、表示层和应用层。该层将应用程序连入网络，TCP/IP协议提供标准的Socket接口使应用程序能够快捷地访问网络。

7.1.3 其他基本概念

1. 局域网和广域网

网络通常可以分为两类：局域网和广域网。

局域网是一种小范围的共享网络，在局域网内计算机通过一种共享介质，通常是电缆来进行局域通信。由于使用共享介质，连接到局域网的计算机按一定顺序发送数据。当一台计算机传送数据时，它独占局域网的使用。比较著名的局域网技术有以太网、LocalTalk、IBM令牌环、FDDI和ATM等。以太网使用总线拓扑结构，多台计算机连接在一根被称为以太（ether）的同轴电缆上。当一个计算机节点检测到以太网空闲时，它试图取得对以太网的控制，取得控制的计算机向网络发送相应的数据。LocalTalk是苹果公司发明的局域网技术。它和以太网类似，不过它的带宽和使用距离要比以太网小得多，当然它的优点是价格十分便宜。IBM令牌环是IBM开发的局域网技术。连接在令牌网上的计算机使用叫做令牌的特殊的短报文来协调环的使用。任何时候环上只有一个令牌。为了发送数据，计算机必须等待令牌到来，然后传输一帧，再向下一台计算机传输令牌。当没有计算机要发送数据时，令牌以高速在网上循环。FDDI也是一种令牌环技术，它使用光纤代替电缆，可以达到100 Mbps的传输速率。ATM是一种高速网络技术。ATM网络由交换机和连接在其上的计算机组成。计算机与ATM交换机之间包含一对光纤，每一根负责一个方向的传输。交换机对光纤上的数据报进行高速交换。由上可见局域网的技术是多种多样的，但是在使用它们时有一个共同的特点：计算机需要特殊的I/O接口设备连接到各类局域网。这一设备被称为网络适配器（network adapter）或者网络接口卡（network interface card）。

广域网是一种大范围连接大量计算机的网络技术。广域网中使用分组交换机进行数据传输。分组交换机是一台特殊的计算机，能够进行高速的I/O操作。分组交换机根据分组的目的地址，向与它相连的特定计算机或分组交换机转发数据分组。常见的广域网技术有ARPANET、X.25、ISDN、帧中继、交换式多兆位数据服务（SMDS）和ATM。和局域网技术相同的是计算机需要特殊的I/O接口网络设备连接到各类广域网。

由上可以看到网络技术是多种多样的，相应地存在各种各样的网络硬件。在计算机系统中操作系统直接负责对各类网络硬件的管理。在各类网络硬件之上是特定的网络驱动程序。类似于磁盘驱动程序，不同的网络硬件设备，它们的驱动程序的实现是截然不同的。对于一个特定的操作系统，网络驱动程序往往实现了特定的网络接口。在这一层之上的操作系统模块以硬件无关的方式对各类网络设备进行访问。比如Novell网络系统，它的网络层和传输层等上层模块使用IPX/SPX协议。通过一个通用的网络接口，IPX协议可以访问各类数据链路层协议，比如以太网、令牌环。

2. 网络通信协议

在因网络硬件而异的数据链路层之上，是相对硬件无关的网络层、传输层和会话层协议。常见的有TCP/IP、IPX/SPX协议。这里值得一提的是不同的操作系统虽然可能实现同一通信协议但实现方法往往不同。比如TCP/IP协议，BSD UNIX有它的实现方法，UNIX SVR4则采用STREAMS实现，Windows2000也有它的实现方法。通常实现了相同的网络通信协议的操作系统上的应用程序可以通过特定的网络编程接口进行相互通信，当然前提是底层硬件可以相互通信。如果一个操作系统只支持IPX/SPX协议，而另一个只支持TCP/IP协议，那么它们之间不可能通过两个协议中的任何一种进行通信，哪怕它们的底层硬件都是以太网卡。由于TCP/IP协议被大部分操作系统所支持，因此就成为互联网通信的标准协议。

3. 网络编程API

通信协议标准通常并不定义应用程序进行网络编程的API，通信协议只是定义了进行计算机网络通信的基本操作，操作系统定义基于通信协议的供上层应用程序的API。套接字API是其中最著名的一个，它被许多操作系统支持。套接字API最初出现在加州大学伯克利分校的BSD UNIX操作系统中，它提供了通过TCP/IP协议进行网络通信的编程接口。对于其他的操作系统，比如Windows操作系统也提供了套接字API。但为了不改变基本的操作系统，套接字API是通过套接字API库来提供的。从开发应用程序的程序员角度来看，套接字库与操作系统直接实现套接字API是相同的，一个使用套接字的应用程序，被移植到另一个操作系统上后只要重新编译一下就可运行。但在实现上，套接字库和操作系统直接提供的本机套接字API是完全不同的。本机套接字API是操作系统的一部分，而套接字库的过程是连接到应用程序并驻留于应用程序地址空间的。当应用程序从套接字库调用过程时，库例程进行一个或多个对操作系统的调用来达到希望的效果。值得注意的是，基本操作系统支持的函数根本不需要与套接字API相像。套接字库的例程向操作系统屏蔽了具体的实现细节，而向应用程序提供了一个套接字接口。

对于目前的商用网络操作系统，比如Windows 2000操作系统，它们通常提供了多种网络编程接口以满足不同的网络应用。在实现上，这些操作系统的网络编程API往往独立于具体的网络协议。如后面提到的NetBIOS编程API，它可以使用TCP/IP、NetBEUI和IPX/SPX协议。这一点和早期的支持网络的操作系统不同，比如早期的BSD UNIX操作系统的套接字API，它是特别针对TCP/IP协议而开发的。把网络编程API和下层的网络协议分开使得现代操作系统的整个网络体系结构更加灵活，以便更好适应网络技术的种类多样性和发展迅速的特点。

4. 域名系统

网络地址通常采用ASCII串来标识。但网络本身的IP地址采用二进制地址，需要一种机制把ASCII串转换成为IP地址。域名服务系统（DNS）负责解决这一问题。当一个应用程序需要解析一个网络地址时，应用程序调用一个库例程。该例程通常通过UDP向本地的DNS服务器发送请求，本地DNS服务器查找到这一网络地址，并把它的IP地址返回给调用者。有了IP地址，应用程序就可以和目的方建立TCP连接，或者向它发送UDP报文。在概念上，因特网被分为许多顶层域，这些域又被分成子域，并不断地被细分。域名的层次由圆点分隔，每个域都控制如何分配它下面的直接子域。DNS的域名空间被划分为不交叉的区域，交给一系列的DNS服务器管理。一台服务器

必须负责有某一后缀的所有计算机。域名系统中的服务器是相互链接的，这样才能使客户通过这些链接找到正确的服务器。每个服务器都知道如何找到根服务器以及那些比它层次低的服务器。为了提高性能，DNS根服务器被复制了许多副本，同时DNS服务器对已经进行过查找的域名进行缓存以提高性能。

7.2 Windows 2000网络体系结构

图7-3描述了Windows 2000的网络构架，从中我们可以看出每个组件是怎样定位于OSI参考模型中，以及哪种网络协议用于层与层之间。OSI模型的层次与网络组件之间的映射关系并不精确，所以存在一些跨层的组件。Windows 2000的网络构架的各类组件包含：

- **网络API** 为应用程序提供一种独立于协议的方式用于网络通信。网络API既能在用户态实现，也可以同时为用户态与核心态实现，并且有时将其他实现一些特定的程序模型或者额外服务的网络API包装在一起（注意：术语**网络API**也可以用来描述一些相关网络软件的编程接口）。
- **传输驱动程序接口（TDI）客户** 是核心态的设备驱动程序，而设备驱动程序通常实现了网络API的核心态部分。TDI客户从发送至协议驱动程序的数据请求（IRP）中获得自己的名称，这些IRP的格式符合Windows 2000传输驱动程序接口标准（可查询DDK的文档资料）。这个标准为设备驱动程序定义了公共编程接口。
- **TDI传送器（TDI transport）** 又称为传送器、NDIS协议驱动程序以及协议驱动程序，是工作在核心态的协议驱动程序。它们接收从TDI客户传来的IRP，然后处理这些IRP中的请求。为了让TDI传送器根据不同的协议（例如TCP，UDP，IPX）将协议头加入IRP的数据中，这一过程可能需要与一个对等实体进行网络通信，而且需要使用NDIS函数（可查询DDK的文档资料）与适配驱动程序通信。通过透明的消息操作，如分段与重组、序列化、确认和重传，TDI传送器简化了应用程序的网络通信。
- **NDIS库（Ndis.sys）** 为适配驱动程序提供了封装，隐藏了Windows 2000核心态环境下的具体细节。NDIS库为适配驱动程序提供支持函数，而且也为TDI传送器的使用提供了函数接口。
- **NDIS小端口驱动程序（NDIS miniport driver）** 是工作在核心态的驱动程序，它负责将TDI传送器接入特定的网络适配器。NDIS小端口驱动程序被封装在Windows 2000 NDIS库

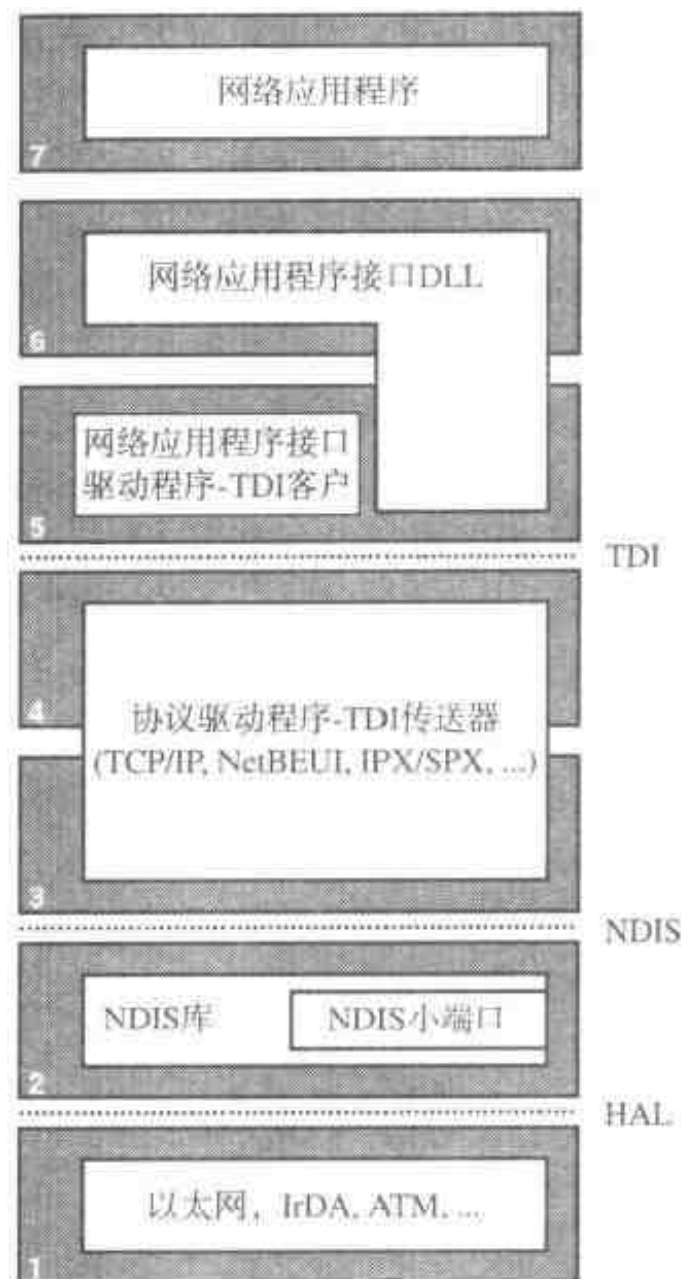


图7-3 OSI模型与Windows 2000网络组件

中。这种封装提供了与微软的Consumer Windows跨平台的兼容性。NDIS小端口驱动程序并不处理IRP，而是将调用表接口注册到NDIS库中，而NDIS库含有指向从库中输出给TDI传送器的函数指针。NDIS小端口驱动程序与网络适配器通信时使用NDIS库函数，这些函数被映射到硬件抽象层（HAL）的函数。

如图中所示，OSI层次并不符合实际软件的设计。例如，TDI传送器常常跨好几层的边界。实际上，软件中的下四层一般统一称为“传输层”。上三层的组件称为“传输使用者”。

在本章的余下部分，我们会讲解图7-3所示的网络组件（有一些是图中没有的），请带着两个问题：这些组件是怎样相互融合的；它们是怎样与Windows 2000联系在一起的。

7.2.1 网络API

Windows 2000实现了多种网络API用于支持传统的应用，以及兼容工业标准。本节将扼要叙述网络API，描述应用程序是怎样使用它们的。我们需要牢记，应用程序使用哪个API取决于API的自身特性，如这个API能够工作在哪些协议之上，它是否支持可靠的通信或者双工，以及它是否允许使用它的应用程序移植到其他Windows平台上。我们将讨论以下所列的网络API：

- 命名管道和邮件槽
- Windows套接字（Winsock）
- 远程过程调用（RPC）
- 公共互连网络文件系统（CIFS）
- NetBIOS

此外，我们会简介一些以上列出的、并且广泛使用于Windows 2000系统的API函数。

1. 命名管道和邮件槽

命名管道（named pipe）和邮件槽（mailslot）是微软起初为OS/2局域网管理器开发的编程API，随后移植到Windows NT。命名管道提供可靠的双向通信，然而邮件槽只提供不可靠的单向通信。邮件槽的一个优点在于它具有广播能力。在Windows 2000中，以上两种API都利用了Windows 2000的安全特性，这样就能让服务器精确地控制哪些客户可以连接它。

名称服务器依照Windows 2000通用命名规范（UNC）为命名管道和客户指定名称，UNC是Windows网络中用于定位资源的，有独立于协议的方法。

(1) 命名管道的操作

命名管道通信由命名管道服务器和命名管道客户组成。命名管道服务器是创建命名管道、让用户连入的应用程序。命名管道的名字格式为\\Server\Pipe\PipeName。其中Server指定了执行命名管道服务器的计算机名（命名管道服务器无法在远程系统上创建命名管道），此计算机名可以是DNS名称（例如mspress.microsoft.com），NetBIOS名称（mspress），或者是IP地址（255.0.0.0）。格式中的Pipe就是字符串“Pipe”，而PipeName是给命名管道指定的唯一名称。此唯一名称可以包含子目录，例如，\\MyComputer\Pipe\MyServerApp\ConnectionPipe。

命名管道服务器使用CreateNamedPipe Win32函数来创建命名管道。函数的输入参数之一是命名管道名字的指针，形式为\\.\Pipe\PipeName。“\\.”是Win32为本地计算机定义的别名。其他

参数则包括一个可选的安全描述符用于保护对命名管道的访问，一个用于指定管道以单向或双向方式工作的标志，一个最大的并发连接数的值，以及一个用于指定管道以字节方式还是消息方式工作的标志。

大多数网络API只以字节方式工作，也就是说，发送方发送的一条消息在接收方可能需要多次接收，然后从碎片中重建完整的消息。以消息方式工作的命名管道简化了接收方的实现，因为一次发送意味着一次接收，它们是一一对应的。因此每次接收完成后，接收方总能得到一条完整的消息，它无需记录前前后后的消息碎片。

以一个特定的名字初次调用CreateNamedPipe创建了这个名字的第一个实例，同时建立了所有与之相关的命名管道实例的行为。如果再次调用CreateNamedPipe，则服务器再创建一个实例，但不能超过第一次调用所指定的最大连接数量。在创建了至少一个命名管道的实例之后，服务器执行ConnectNamedPipe Win32函数，用来让已有命名管道和客户建立连接。CreateNamedPipe既可以同步执行，也可以异步执行，此调用直到客户与实例建立了连接（或错误产生）才算完成。

命名管道客户使用Win32 CreateFile和CallNamedPipe函数连接服务，指定服务器创建的管道名称。如果服务器已执行了ConnectNamedPipe调用，则客户的安全配置文件以及它所请求的对管道的访问权限（读、写）都通过命名管道安全描述符进行验证。如果客户被授予访问命名管道的权限，它就会收到一个代表命名管道连接的客户端句柄，此时服务完成对ConnectNamedPipe的调用。

在建立命名管道的连接之后，客户和服务端可以使用ReadFile和WriteFile Win32函数从管道中读取和写入管道。命名管道支持同步和异步的传输操作。图7-4表示了服务器与客户之间通过命名管道的通信。

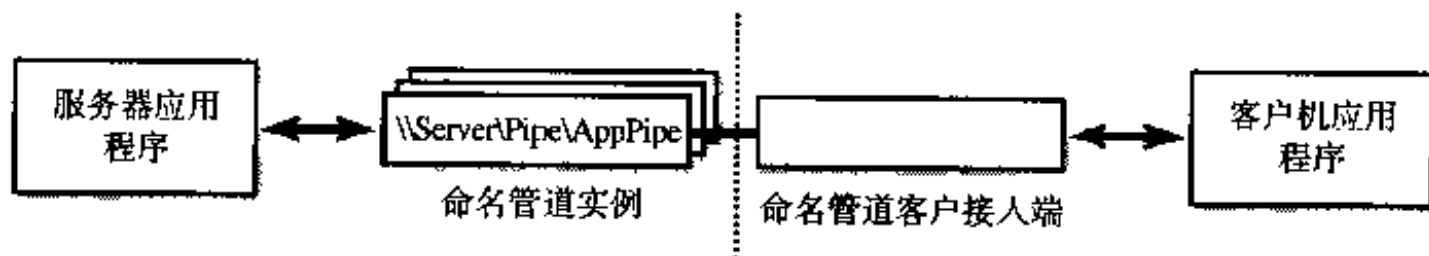


图7-4 命名管道的通信

命名管道的网络API有个特点，就是它允许服务器通过使用ImpersonateNamedPipeClient 函数扮演客户的角色。

(2) 邮件槽的操作

邮件槽提供一种不可靠的、单向广播机制。例如，使用这种通信方式的应用程序可以是时间同步服务，它可以每几秒就向域内广播源端时间。然而，对于网络中的每台计算机并不都需要接收源端时间的消息，因此，使用邮件槽是客户端网络应用的补充。

邮件槽像命名管道一样，与Win32 API集成在一起。邮件槽服务器用CreateMailslot函数创建一个邮件槽。CreateMailslot名称的输入格式为“\\Mailslot\\MailslotName”。其中，邮件槽服务器只能在执行它的机器上创建邮件槽，它的名称能包含子目录，这一点与命名管道相似。CreateMailslot也可以接收安全描述符用于控制客户对邮件槽的访问。CreateMailslot返回的句柄会被重叠，即用此函数返回的句柄进行的操作是异步执行的，比如发送和接收消息。

由于邮件槽是单向而且不可靠的，CreateMailslot并不像CreateNamedPipe需要很多参数。在建立了邮件槽之后，服务器仅仅监听到达的客户消息，这一过程使用ReadFile函数和此邮件槽的句柄。

邮件槽客户使用类似于命名管道的名称格式，不过针对指定域的广播，它作了相应的改动。为了向一个指定的邮件槽实例发送一条消息，客户调用CreateFile时指定了计算机名字。例如“\\Server\Mailslot\MailslotName\”。（客户端可以使用“\\.”指定本地计算机。）如果客户所在域中有多个同名的邮件槽服务，但是分布在不同的计算机上，此时客户想要得到一个统一的句柄用来对这些邮件槽进行广播，那它可以以格式“*\Mailslot\MailslotName”来指定服务器名称。当客户不在此域中，则可以以格式“\\Domain\Mailslot\MailslotName”进行广播。

在获得客户端的邮件槽句柄之后，客户可以调用WriteFile发送消息。但是邮件槽对广播消息有长度小于425字节的限制。如果一个消息大于426字节，邮件槽就使用点对点的可靠的通信机制，这也就等于失去了广播能力。邮件槽的另一个奇特之处是，长度为425或426字节的消息会截短为424字节。所以，Windows 2000并不支持425和426字节的邮件槽消息。图7-5说明一个客户对多个邮件槽服务器进行广播的方式。

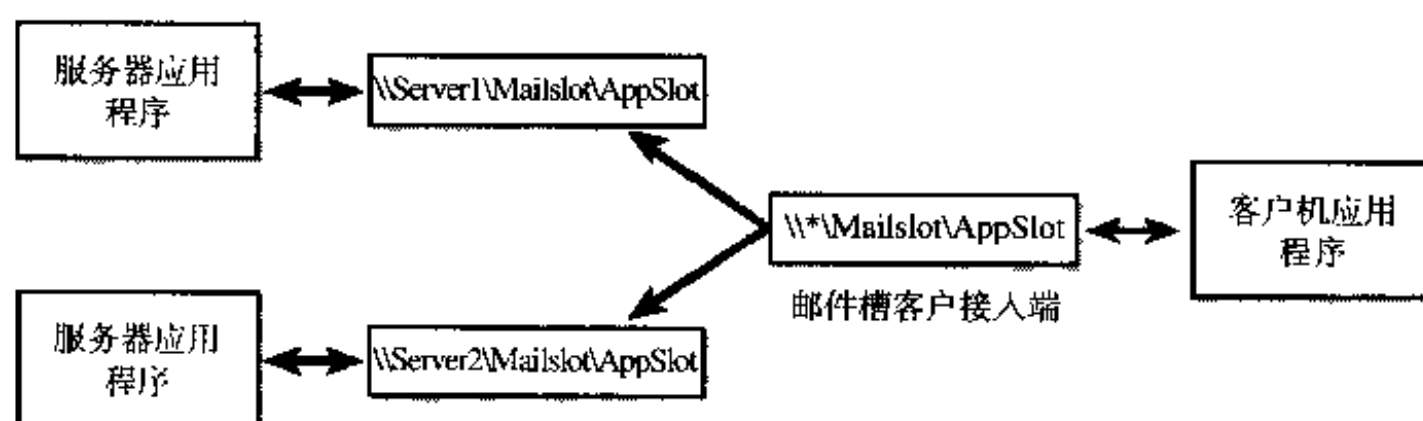


图7-5 邮件槽的广播

(3) 命名管道和邮件槽的实现

正是由于命名管道和邮件槽与Win32紧密地结合在一起，因此它们都实现在Kernel32.dll的Win32客户端DLL中。应用程序使用ReadFile和WriteFile函数来发送和接收命名管道和邮件槽的消息，这些函数是基本的Win32 I/O例程。用于创建CreateFile函数也是标准的Win32 I/O例程。然而，由命名管道和邮件槽指定的名称确定了由命名管道文件系统驱动程序（\Winnt\System32\Npfs.sys）和邮件槽文件系统驱动程序（\Winnt\System32\Drivers\Msfs.sys）所管理的系统名字空间（如图7-6所示）。命名管道文件系统驱动程序创建了一个名为\Device\NamedPipe的设备对象和一个相应的符号连接\??\Pipe，邮件槽文件系统驱动程序创建了名为\Device\Mailslot的设备对象和它的符号连接\??\Mailslot。以\\.\Pipe\...和\\.\Mailslot\...为格式的名称，这些名称通常传给CreateFile，它们都带有前缀\\.\。这些前缀会被译成\??\，这样，名称可以通过设备对象的符号连接进行解析。CreateNamedPipe和CreateMailslot是两个比较特别的函数，它们使用相应的本地函数NtCreateNamedPipeFile和NtCreateMailslotFile。

在本章的稍后部分，我们会讨论当一个指定了远程命名管道或邮件槽的名字解析成远程系统时，包含文件系统驱动程序在内的重定向器所起到的作用。然而，当服务器创建了一个命名管道或邮件槽，或者当它们被客户打开时，此机器上相应的文件系统驱动程序（FSD）会被调用。

FSD在核心态实现命名管道和邮件槽的主要原因是，它们集成在对象管理器的名字空间中，而且能够使用文件对象来表示已打开的命名管道和邮件槽。这种集成给我们带来以下几种好处：

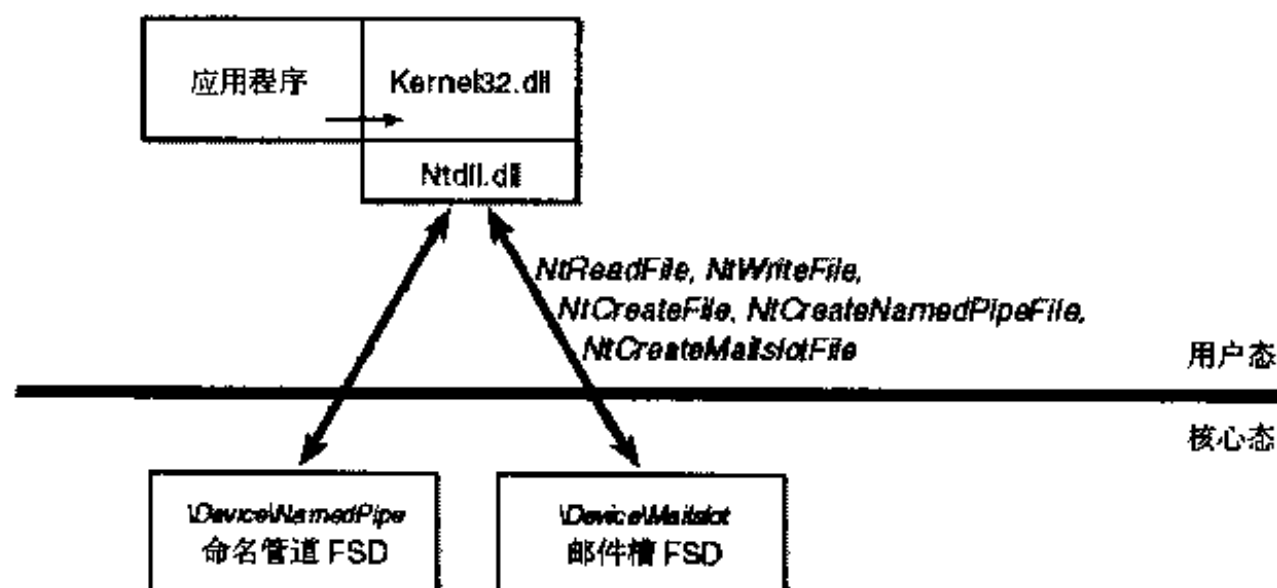


图7-6 命名管道和邮件槽的实现

- FSD使用核心态安全性函数为命名管道和邮件槽实现标准的Windows 2000安全性。
- 应用程序能够使用CreateFile打开命名管道或邮件槽，因为FSD与对象管理器的名字空间集成在一起。
- 应用程序能够使用Win32函数，如ReadFile和WriteFile，与命名管道和邮件槽交互。
- FSD依赖于对象管理器对句柄和引用计数的跟踪，而这些句柄和引用计数属于命名管道和邮件槽的文件对象。
- FSD利用原有的子目录特性，可以实现它们自身的命名管道和邮件槽。

因为命名管道和邮件槽名字解析使用重定向器FSD进行跨网络通信，它们间接依赖CIFS协议（后面会讲述）。CIFS使用IPX，TCP/IP和NetBEUI协议，所以只要应用程序支持这些协议之一就可以使用命名管道和邮件槽。

2. Windows套接字

Windows套接字（Winsock）是微软根据BSD套接字而实现的，套接字是一类编程API，它自20世纪80年代起成为UNIX系统在因特网上的通信标准。Windows 2000对套接字的支持使UNIX网络应用移植到Windows 2000上变得相当快捷。Winsock包括大多数BSD套接字的功能，同时也增加了一些带有微软特色的增强功能，而这些功能又在不断地改进。Winsock既支持不可靠的、面向无连接的通信，也支持可靠的、面向连接的通信。Windows 2000提供Winsock 2.2，它可以包含在或者作为附加软件安装在Consumer Windows的所有版本中。

Winsock包含以下几个特点：

- 支持拆分重组和异步模式的应用程序I/O操作。
- 定义了服务质量（QoS）的标准。当下层网络支持QoS的时候，应用程序可以协商延迟和带宽要求。
- 扩展性。除了必须支持Windows 2000所指定的协议以外，Winsock还能使用其他协议。
- 除了那些由Winsock应用程序所使用的协议定义的名字空间以外，它还支持集中的名字空

间。一个服务器可以在活动目录内发布它的名称，例如，使用名字空间扩展，客户可以在活动目录内查找服务器的地址。

我们先了解一下Winsock的一些典型操作，然后描述Winsock的一些扩展途径。

(1) Winsock的操作

调用初始化函数，完成对Winsock API的初始化之后，Winsock应用的第一步就是创建一个套接字，它表示一个通信端点。一个套接字必须绑定到本机地址，因此绑定是应用操作的第二步。Winsock是一种独立于协议的API，所以它的地址可以指定为安装在系统上的任何协议的地址，当然，此系统所安装的协议是Winsock可操作的。在绑定完成之后，服务器和客户的执行步骤有所不同，而且对应面向连接和面向无连接的套接字操作步骤也不一样。

一个面向连接的Winsock服务器在套接字上执行listen操作，指明此套接字可以支持的连接数量。然后，再执行accept操作，让客户连接套接字。如果有一个等待的连接请求，accept调用可以立即返回；否则，只有当连接请求到达时该调用才能完成。当一个连接已经创建之后，accept函数返回一个新的套接字，它代表此连接的服务器端。此服务器可以使用诸如recv和send函数来执行发送和接收操作。

面向连接的客户使用Winsock的connect函数连接到服务器，此函数需要指定一个远程地址。当一个连接建立的时候，客户可以在它的套接字上发送和接收消息。图7-7说明了Winsock客户与服务器之间的面向连接的通信。

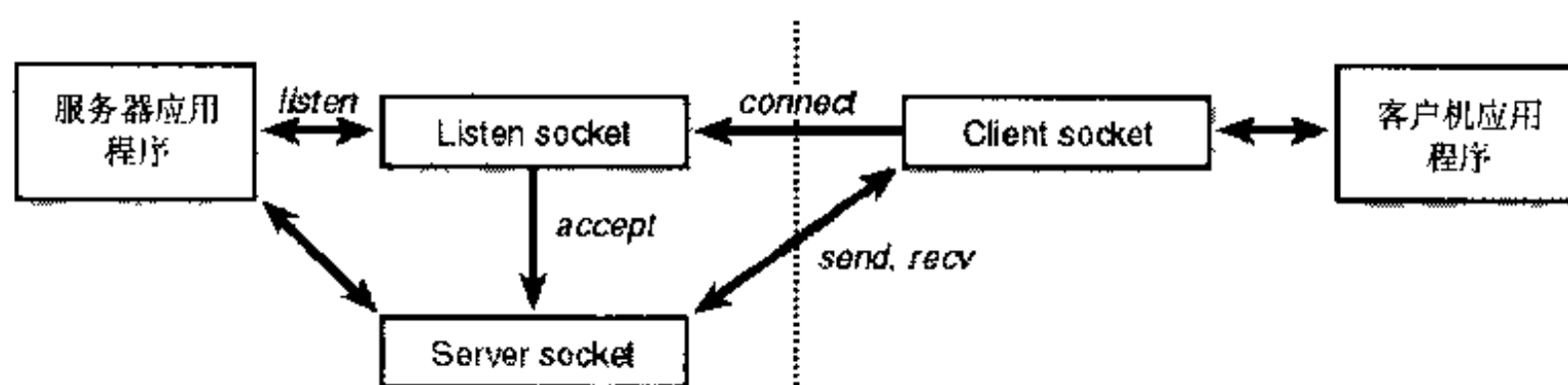


图7-7 面向连接的Winsock操作

而面向无连接的服务器在绑定地址之后，与面向无连接的客户端没有任何区别：它只需在每条消息中指定远端的地址便可以发送和接收消息。当使用无连接的消息时，我们称这种消息为数据报，发送方通过下一次接收操作所返回的错误码，可以得知这次发送的消息对方没有收到。

(2) Winsock的可扩展性

一种Windows编程观点认为Winsock API与Windows消息集成在一起是一大特点。Winsock应用可以利用这一特性执行异步套接字操作，以及通过标准Windows消息或回调函数的执行来接收操作完成时的通知。这一功能简化了Windows应用的设计，因为这一类应用不必是多线程或者管理同步对象，这一特性将不再需要执行网络I/O操作与从窗口管理器响应用户输入及请求来刷新用户窗口。基于消息版本的BSD风格的Winsock函数名称一般以前缀WSA开头，例如，WSAAccept。

除了支持BSD套接字的函数，微软还添加了少许非Winsock标准的函数。其中的两个，AcceptEx和TransmitFile值得讨论一番，因为许多Windows 2000的Web服务器使用这两个函数来

获取高性能。AcceptEx是accept函数的一个版本，前者返回客户的地址和第一条消息，而后者仅仅处理与客户建立连接。使用AcceptEx函数，Web服务器可以避免执行多次Winsock函数，相反，使用accept就不行。

在与客户建立了连接之后，Web服务器通常发送一个文件给客户，例如网页。由于TransmitFile函数的实现与Windows 2000缓存管理器集成在一起，因此客户能够直接从文件系统的缓存中发送一个文件。以这种方式发送一个文件称之为零拷贝，因为服务器没有必要为了发送文件而接触它自身，服务器只需指定一个指向文件的句柄和所传数据的范围。另外，TransmitFile允许一个服务器在文件之前预先挂起数据或者在文件之后追加数据，这样服务器就可以发送头部信息，这些头部信息包含Web服务器的名称，以及一个指明消息大小的域。Windows 2000自带的IIS 5.0既使用AcceptEx，又使用TransmitFile。

(3) 扩展Winsock

在Windows 2000上，Winsock是一种可扩展的API，因为第三方厂商可以添加传输服务提供者将其他协议接入Winsock，另外还有名字空间服务提供者支持Winsock名字解析工具。利用Winsock的服务提供者接口（SPI）将服务提供者接入Winsock。当传输服务提供者在Winsock中注册后，Winsock利用传输服务提供者实现套接字函数，它针对提供者的地址类型实现，例如connect和accept。对于传输服务提供的实现没有任何限制，但是一般包括核心态中与传输驱动程序之间的通信。

任何一个Winsock的客户/服务器应用程序都要求服务器的地址信息对客户来说总是可以得到的，这样客户就能连接到服务器。运行在TCP/IP上的标准服务器使用一些众所周知的地址，这样客户很容易连接这些服务器。只要浏览器知道运行Web服务器的计算机名称，它就能通过指定公开的Web服务器地址（IP地址加上:80，80是HTTP的端口号）连接Web服务器。名字空间服务提供者使某些服务器能以其他方式登记它们的名称。例如，在服务器端，一个名字空间提供者可以在活动目录中登记服务器的地址，而在客户端，它可以在活动目录内查询服务器的地址。名字空间服务提供者实现了标准Winsock名字解析函数，如gethostbyaddr，getservbyname和getservbyport，使用这些函数使得上述功能成为现实。

(4) Winsock的实现

Winsock的实现如图7-8所示。它的应用程序接口由一个API DLL组成，Ws2_32.dll（\Winnt\System32\Ws2_32.dll），它为应用程序提供Winsock函数Ws2_32.dll调用名字空间和传输服务提供者实施名称或消息操作。Msafd.dll库作为微软支持Winsock协议的传输服务提供者，它利用Winsock Helper库与核心态的协议驱动程序进行协议细节相关的通信。例如，Wshtcpip.dll是TCP/IP Helper，Wshnetbs.dll是NetBEUI Helper。微软Winsock的扩展函数由Mswsock.dll（\Winnt\System32\Mswsock.dll）实现，如TransmitFile，AcceptEx以及WSARecvEx等。Windows 2000附带TCP/IP、NetBEUI、AppleTalk、IPX/SPX、ATM以及IrDA（红外数据协会）的Help DLL，还有DNS（TCP/IP）、活动目录和IPX/SPX的名字空间服务提供者。

如同命名管道和邮件槽API，Winsock与Win32 I/O模型集成在一起，并且利用文件句柄表示套接字。这种特性得益于核心态的文件系统驱动程序的帮助，因此Msafd.dll使用辅助函数驱动程序（AFD - \Winnt\System32\Drivers\Afd.sys）实现基于套接字的函数。AFD是一个TDI客户，它

通过发送TDI IRP到协议驱动程序来执行网络套接字操作，例如发送和接收消息。AFD并不是针对某一种特定的协议驱动程序设计；然而，Msafd.dll告诉AFD每个套接字所使用的协议名称，这样，AFD就能够打开代表协议的设备对象。

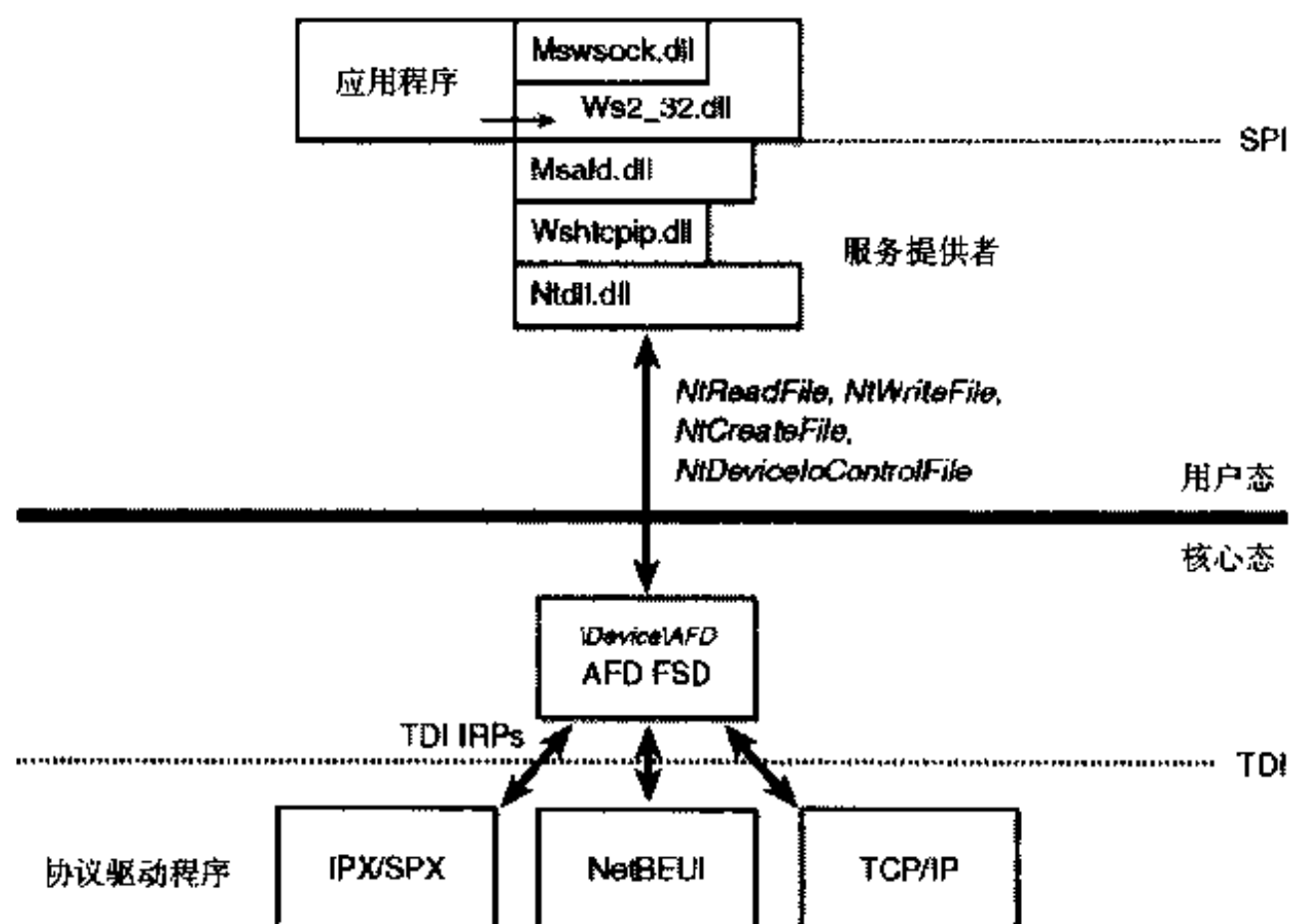


图7-8 Winsock的实现

3. 远程过程调用

远程过程调用（RPC）是一种网络编程标准，它起源于20世纪80年代早期。开放式软件基金会（OSF，现在是开放式团体）使RPC成为分布式计算环境（DCE）的分布式计算标准。虽然SunRPC是另一个RPC的标准，但是微软RPC的实现兼容OSF/DCE标准。RPC建立在其他的网络API上，这样就能提供另一类编程模型，它将应用程序开发人员从网络编程的细节中解脱出来，即它隐藏了网络的实现。

(1) RPC的操作 RPC工具让程序员能创建任一个包含若干过程的应用，其中一些过程可以本地执行，而另一些可以跨网络远程执行。它提供一种网络操作的过程式视角，而不是以传输为中心视角，这样也就简化了分布式应用的开发。

传统网络软件的结构以I/O处理模型为主。例如，在Windows 2000里，应用程序对远程I/O的请求会产生网络操作。操作系统将请求转发到重定向器上，重定向器作为远程文件系统，使客户与远程文件系统的交互对客户是不可见的。重定向器将操作传给远程文件系统，远程文件系统完成请求并返回结果之后，本地网卡产生中断。随后，核心处理中断，原先激发的I/O操作至此已全部完成，然后返回结果给调用者。

RPC的操作方式却完全不同。就像其他结构化应用程序一样，RPC程序有自己的主程序，它调用过程或过程库来执行一些具体任务。RPC应用与常规应用的差别在于RPC应用中所使用的一

些过程库在远程计算机上执行，而其他应用在本地执行过程，如图7-9所示。

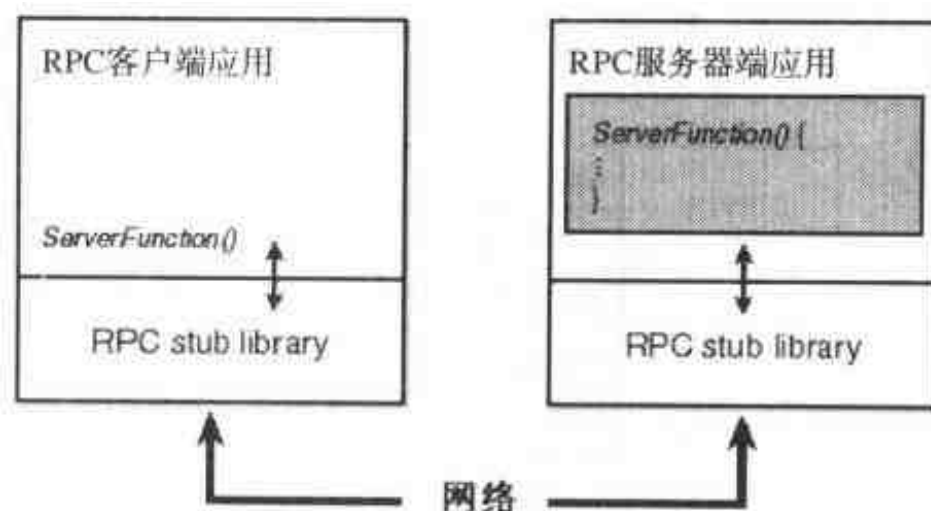


图7-9 RPC的操作

对于RPC应用程序，所有过程在形式上都是本地执行的。换句话说，程序员无需编写任何代码来传输跨网络的计算机或I/O相关的请求，以及选择网络协议，处理网络错误，等待结果等。RPC软件自动完成了这些工作。Windows 2000 RPC工具能够在系统中加载的任何传送器之上操作。

为了编写RPC应用，程序员决定哪些过程需要本地执行，哪些需要远程执行。例如，我们假设一个普通的工作站与Cray超级计算机或者专用的高速向量机之间存在网络连接，如果程序员编写一个要进行大矩阵群的操作，那么将数学计算从本机通过RPC操作移至远程计算机对于平衡性能负载是非常有意义的。

RPC的操作以如下方式工作：当应用程序运行时，它调用本地和远程过程。对后者来说，应用程序连接到本地包含桩过程的静态连接库或者DLL，每个桩过程对应一个远程过程。对一些简单应用，桩过程与应用静态连接，而对一些大组件，桩过程都包含在独立的DLL中。在DCOM中，后者的连接方式被广泛地采用。桩过程与远程过程有相同的名称，使用相同的接口，然而，桩过程并不执行所要求的操作，它只是将应用程序传入的参数进行参数打包，然后再送至网络上传送。所谓参数打包就是将参数排序后以一种特殊的方式打包成适应网络连接的格式，例如解析引用，挑选指针所引用的数据结构的一个拷贝等。

桩过程调用RPC运行时的过程来定位远程过程所寄居的计算机，确定远程计算机使用哪种传送机制，然后利用传输软件发送请求给它。当远程服务器收到RPC请求之后，它对参数解包（参数打包的逆操作），重建原有的过程调用，然后调用此过程。当服务器完成调用之后，它执行相反的操作将结果返回调用者。

除了上述提到的基于同步函数调用的接口以外，Windows 2000 RPC还支持异步RPC。异步RPC使RPC应用程序执行一个函数却不必要等它执行完成，就可以继续应用程序的运行。而应用程序在取得来自服务器的应答之后，可以在之后的其他代码中执行相应的程序，RPC运行库产生一个事件对象，客户通过它与异步调用相关联。客户能够使用标准的Win32函数探测函数是否完成，如WaitForSingleObject。

除了RPC运行库以外，微软的RPC工具还有一个编译器，叫做微软接口定义语言（MIDL）编译器。MIDL编译器简化了RPC应用程序的开发。程序员只需编写一系列常规函数原型（假定是C或C++应用）用于描述远程例程，然后将这些原型存入一个单独的文件。随后，程序员给原

型添加了一些额外的信息，如在网络中的唯一标识，它用于例程的打包以及确定版本号，然后再加上一些属性指明参数是输入、输出还是两者都是。这些经过修饰的原型组成了开发人员的接口定义语言文件。

一旦IDL文件创建完毕，程序员用MIDL编译器编译此文件，生成了客户端和服务端的桩例程，另外还有一些在应用程序中使用的头文件。当客户端应用连接到桩例程文件的时候，所有的远程过程引用被解析。然后远程过程利用相似的过程在服务器上进行安装。程序员调用已有的RPC应用程序时只需编写客户端软件，然后将应用程序连接到本地的RPC运行时工具就行了。

RPC运行库使用RPC传输提供者接口与传输协议对话，提供者接口作为RPC工具与传送器之间的度层，它将RPC操作映射到传送器提供的函数。Windows 2000 RPC工具为命名管道、NetBIOS和TCP/IP实现了传输提供者DLL。你可以编写新的提供者DLL支持其他传送器。RPC工具有一点比较赶时髦，就是它被设计成支持不同的网络安全设施。

大多数Windows 2000的网络服务器是RPC应用程序，即本地进程和远程计算机的进程都可以调用它们。这样，远程客户计算机可以调用服务器服务来列出共享清单，打开文件，写入打印队列，或在你的服务器上激活用户，或者它能够调用信使服务器给你直接发送消息等（当然，所有这些操作都取决于安全性限制）。

服务器名称发布，是一个服务器在客户能够查询的位置进行注册名称的能力，它属于RPC并且与活动目录集成在一起。如果活动目录没有安装，RPC名称定位器服务器将退回到NetBIOS广播。这种行为保证了与Windows NT4的交互操作，而且使RPC在独立的服务器和工作站上也能发挥作用。

(2) RPC的安全性

Windows 2000 RPC与安全支持提供者（SSP）集成在一起，服务器可以使用验证或加密的通信。当RPC服务器要建立安全通信，它必须在SSP内登记SSP相关的主名称。当客户绑定到一个服务器时，先登记它的安全证书，然后指定服务器的主名称。在绑定的时候，客户还指定了它所要的验证级别。各种不同的验证级别保证了只有经过授权的客户才能连接到服务器，确认服务器收到的任何一条来自授权用户的消息，校验RPC消息的完整性，看其是否被操作过，甚至对RPC消息的数据进行加密。显然，验证级别越高，处理开销越大。

SSP处理网络通信验证与加密的一些细节问题，除了RPC以外还有Winsock。Windows 2000包括大量内置的SSP，其中有Kerberos SSP实现了Kerberos 5验证版本，还有Secure Channel（SChannel）实现了Secure Sockets Layer（SSL），Transport Layer Security（TLS）协议，以及私有通信技术（PCT）。如果没有指定SSP，RPC软件将使用命名管道的内置安全性。

RPC安全性的另一个特点就是，利用RpcImpersonateClient函数，服务器就有扮演客户安全性标识的能力。在完成扮演客户的操作之后调用RpcRevertToSelf或者RpcRevertToSelfEx，它恢复为自己的安全性标识。

(3) RPC的实现

图7-10描绘了RPC的实现方式，一个基于RPC的应用程序与RPC运行时DLL相连接（\Winnt\System32\Rpcrt4.dll）。除了提供相应的发送、接收打包数据的函数，RPC运行时DLL还为应用程序的RPC函数桩提供参数打包和参数解包的函数。RPC运行时DLL包含一些例程支持网络RPC的处理，而不仅仅是本地RPC的形式。本地RPC可以使在同一个系统上的两个进程进行通

信，而且RPC运行时DLL使用核心态的本地过程调用（LPC）工具作为本地网络API。当RPC是基于非本地通信机制的时候，RPC运行时DLL使用Winsock、命名管道或者消息队列API。

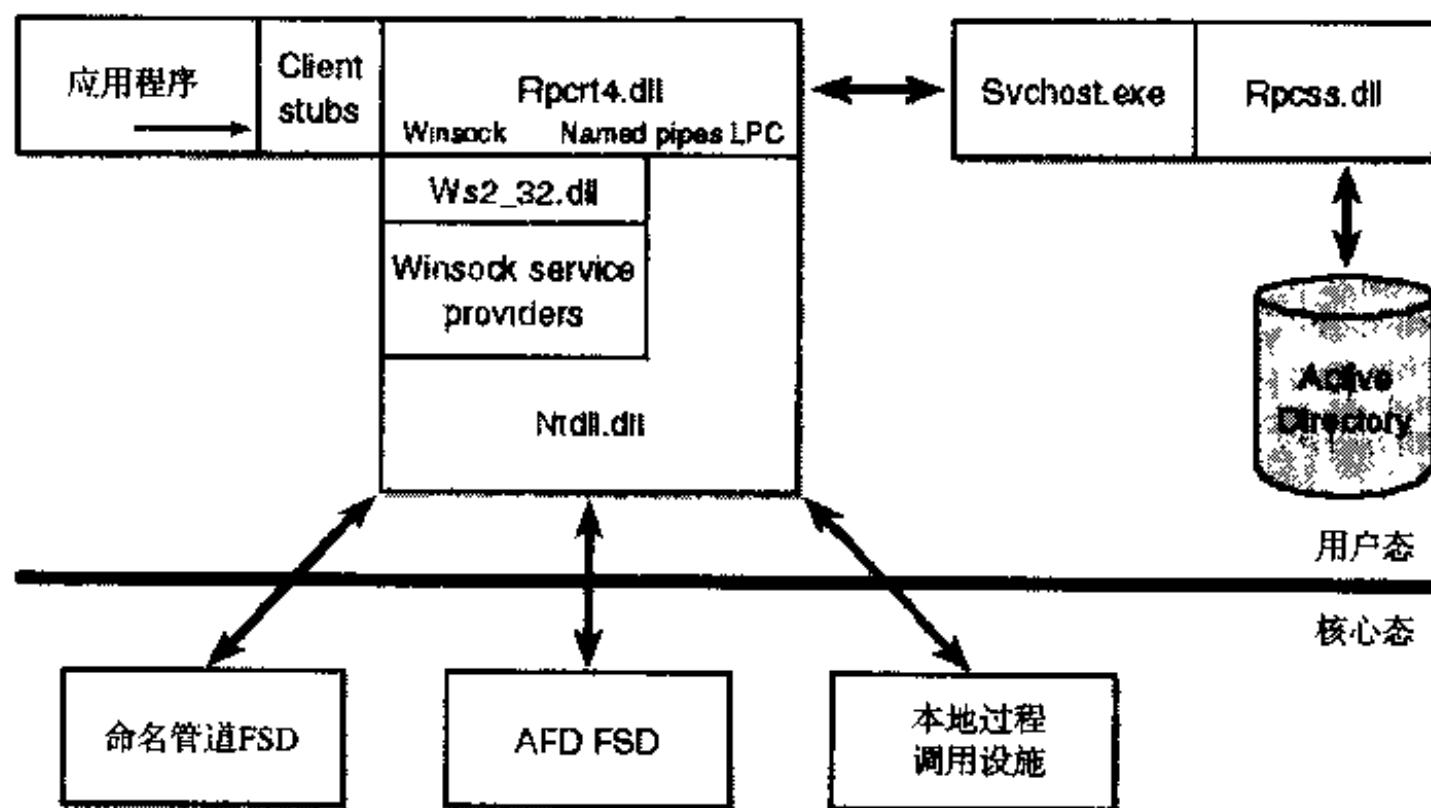


图7-10 RPC的实现

对名称注册和查询来说，RPC应用程序与RPC名称服务DLL连接（\Winnt\System32\Rpcns4.dll）。此DLL与RPC子系统（RPCSS - \Winnt\System32\Rpcss.dll）进行通信，RPC子系统是以Win32的服务的形式实现的。RPCSS本身也是一个RPC应用程序，它与其他子系统上的RPCSS通信，处理名称查询和注册（有一点要澄清的是，图7-9没有表示RPCSS与RPC运行时DLL的连接）。

4. 通用互联网文件系统

通用互联网文件系统（CIFS）作为服务器消息块（SMB）协议的增强形式，是Windows 2000用于互联网文件共享的协议。因为应用程序通过标准的Win32文件I/O函数访问远程文件，所以应用程序不直接使用CIFS协议，但是CIFS负责处理这种远程I/O调用请求。CIFS定义了打印机共享规范，因此Windows 2000自身也使用CIFS进行打印机共享。尽管CIFS自身不是一个API，我们在这里提到它是因为文件和打印机共享建立在CIFS基础之上，同时它通过Win32 API向应用程序展现。

CIFS作为一个公开的微软标准（在Platform SDK中有相关文档）允许第三方与Windows 2000文件服务器以及Windows 2000文件共享客户之间进行互操作。比如Samba共享软件套件使UNIX操作系统可以向Windows 2000客户提供文件服务，同时使UNIX环境的应用程序可以访问Windows 2000上的文件。其他支持CIFS的平台还有DEC VMS和Apple Macintosh。

Windows 2000上的文件共享基于重定向器FSD。重定向器FSD运行在客户机上，它与运行在服务器上的服务器FSD进行通信。重定向器FSD截获指向服务器上文件的Win32文件I/O调用请求，向服务器的文件系统发送相应的CIFS消息去执行客户端的请求。图7-11显示了重定向器FSD和服务FSD的互相通信。

重定向器FSD和服务FSD与Windows 2000的I/O系统是整合在一起的，这样的实现比起在

用户空间的文件服务器实现有很多优点：

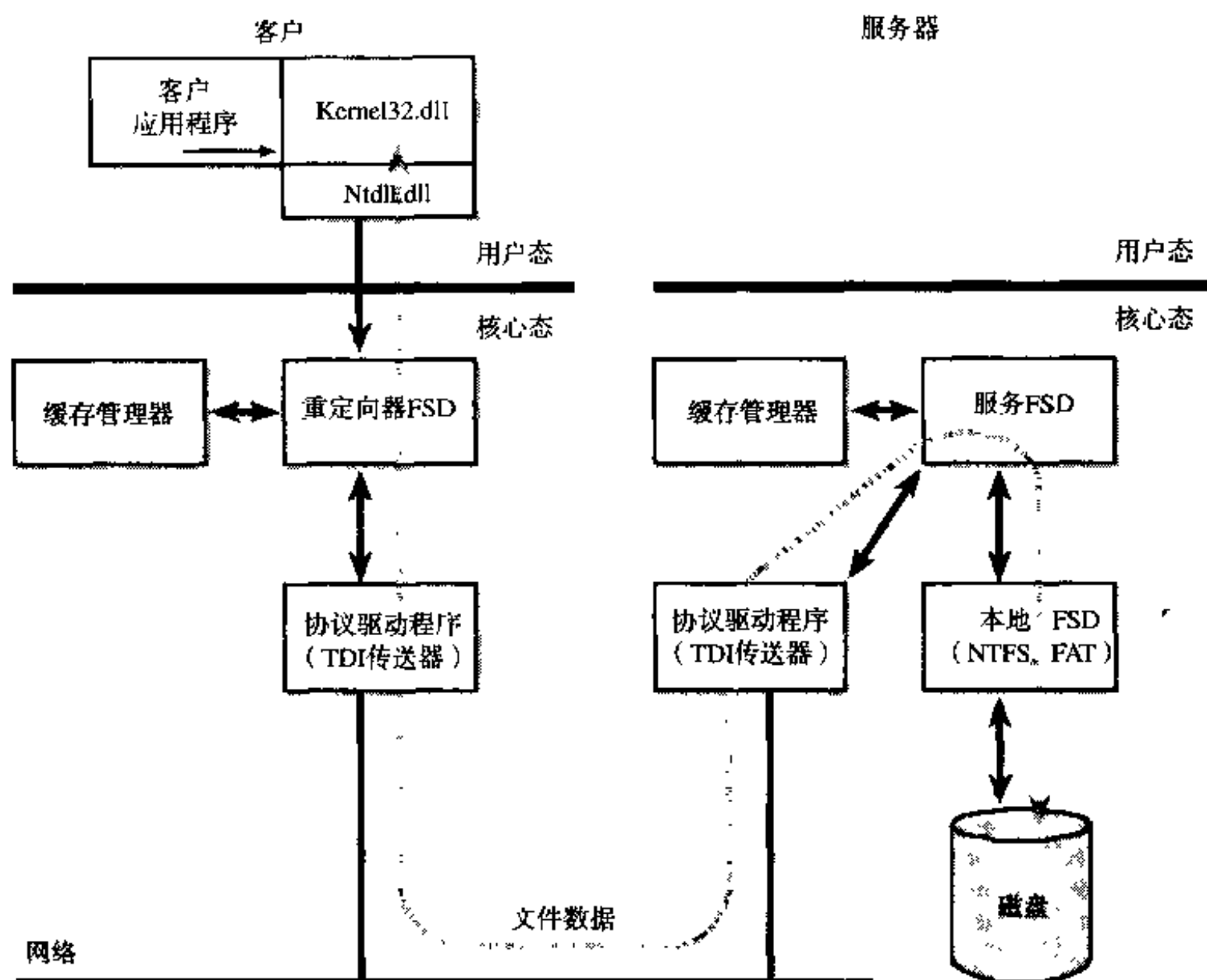


图7-11 CIFS文件共享

- 它们可以直接与TDI传输口和本地FSD交互。
- 它们可以和缓存管理器整合在一起，在客户端无缝地缓存服务器上的文件。
- 应用程序可以使用标准的Win32文件I/O函数访问远程文件，比如CreateFile、ReadFile和WriteFile。

Windows 2000的重定向器FSD和服务FSD遵从标准网络资源命名规范，所有核心态的文件服务器和客户端均遵从这一规范。如果一个远程文件共享用一个驱动器号来连接，网络文件的命名就和本地文件相同。此外，重定向器FSD还支持UNC命名。网络资源的名字解析将在本章的“网络资源的名字解析”部分介绍。

(1) CIFS的实现

和许多其他设备驱动类型一样，Windows 2000的重定向器FSD采用端口/小端口模型。微软提供了名为\Winnt\System32\Drivers\Rdbss.sys的重定向器FSD端口库。开发者可以针对该库编写自己的小端口驱动程序。重定向器FSD端口库隐藏了实现一个重定向器FSD的许多细节，比如重定向器FSD和缓存管理器、内存管理器、TDI传输口的整合。CIFS小端口驱动程序的文件名为

\Winnt\System32\Drivers\Mrxsmb.sys。

Mrxsmb.sys使用了前面几章讲述的缓存管理器来缓存文件数据并进行智能预读。CIFS的小端口驱动程序通过TDI API向远程服务器发送CIFS命令。它可以使用任何支持TDI API的传输方式，比如NetBEUI、NetBT（基于TCP/IP的NetBIOS）和TCP/IP。

在充当文件服务器角色的系统上，服务器FSD（\Winnt\System32\Drivers\Srv.sys）监听发自客户机的CIFS命令，同时作为代理接口访问服务器本地FSD。通过本地Windows 2000 FSD实现的文件系统接口，服务器FSD实现了零拷贝发送的能力。文件系统接口允许服务器FSD获得描述一个内存描述子列表（MDL）。这个列表描述了暂存在文件系统缓存的文件数据。服务器FSD可以把MDL传给TDI传输口，TDI传输口使用网络适配器驱动程序进行网络传输。如果没有本地FSD和缓存管理器的支持，服务器FSD就把文件数据拷贝到自己的缓存中，随后把它传给TDI传输口。

服务器FSD和重定向器FSD有相应的Win32服务：服务器服务和工作站服务。这些服务运行在服务控制管理器（SCM）进程中，它们提供了对相应驱动程序的管理接口。

(2) 分布式文件缓存

如果只有一个客户访问服务器的某一文件，显然在客户端缓存文件数据是安全的。当两个客户同时访问一个文件时，必须采取一定的措施保证客户和服务器的数据文件的一致。Windows 2000采用机会锁（oplock）的机制来解决分布式缓存的一致性（distributed cache coherency）问题。当一个客户访问服务器文件时，它必须首先请求机会锁。服务器授予客户的机会锁的类型决定了客户能采用的缓存方式。有三种主要的机会锁类型：

- **I级机会锁** 用于一个客户独占一个文件。拥有这种类型机会锁的客户能在客户端对相应文件进行读写缓存。
- **II级机会锁** 表示文件共享锁。拥有II级机会锁的客户可以对相应文件进行缓存读，但是任何对文件的写操作会使II级机会锁失效。
- **批处理机会锁** 是有最大许可权的机会锁。拥有这种类型机会锁的客户不仅能在客户端对相应文件进行读写缓存，同时打开关闭文件不需要额外请求机会锁。批处理机会锁通常用来支持批处理文件的执行，在执行时通常需要反复打开关闭一个文件。

如果一个客户没有机会锁，那么它就不能在本地进行缓存读或缓存写，而是直接从服务器接收数据或者把所做的所有修改直接发往服务器。

举一个例子来帮助说明机会锁的操作，如图7-12所示。当第一个客户访问服务器的文件时，服务器自动授予I级机会锁。客户的重定向器FSD可以在客户的文件系统缓存中缓存对文件读操作和写操作的数据。如果第二个客户需要打开这个文件，它也请求I级机会锁。但此时由于有两个客户访问同一文件，服务器必须采取一定的措施保证两客户数据文件的一致。如果第一个客户已经对文件进行写操作，如图7-12所示，服务器收回I级机会锁同时不授予任何客户机会锁。当第一个客户的机会锁被收回，或者说被打破（broken）时，第一个客户把所有缓存的文件数据写回服务器。

如果第一个客户没有写过文件，那么第一个客户的I级机会锁会被打破成II级机会锁，同时也授予第二个客户II级机会锁。这时两个客户都只能缓存读操作数据。如果此后有任何一个客户对文件进行写操作，服务器会回收它们的机会锁，那时只能进行无缓存的操作。对于一个文件的打开实

例，一旦机会锁被打破，它们将不能被重新获得。但是如果客户关闭文件再重新打开它，服务器会根据打开此文件的客户数量和是否有--一个客户对文件进行写操作来决定授予客户何种级别的机会锁。

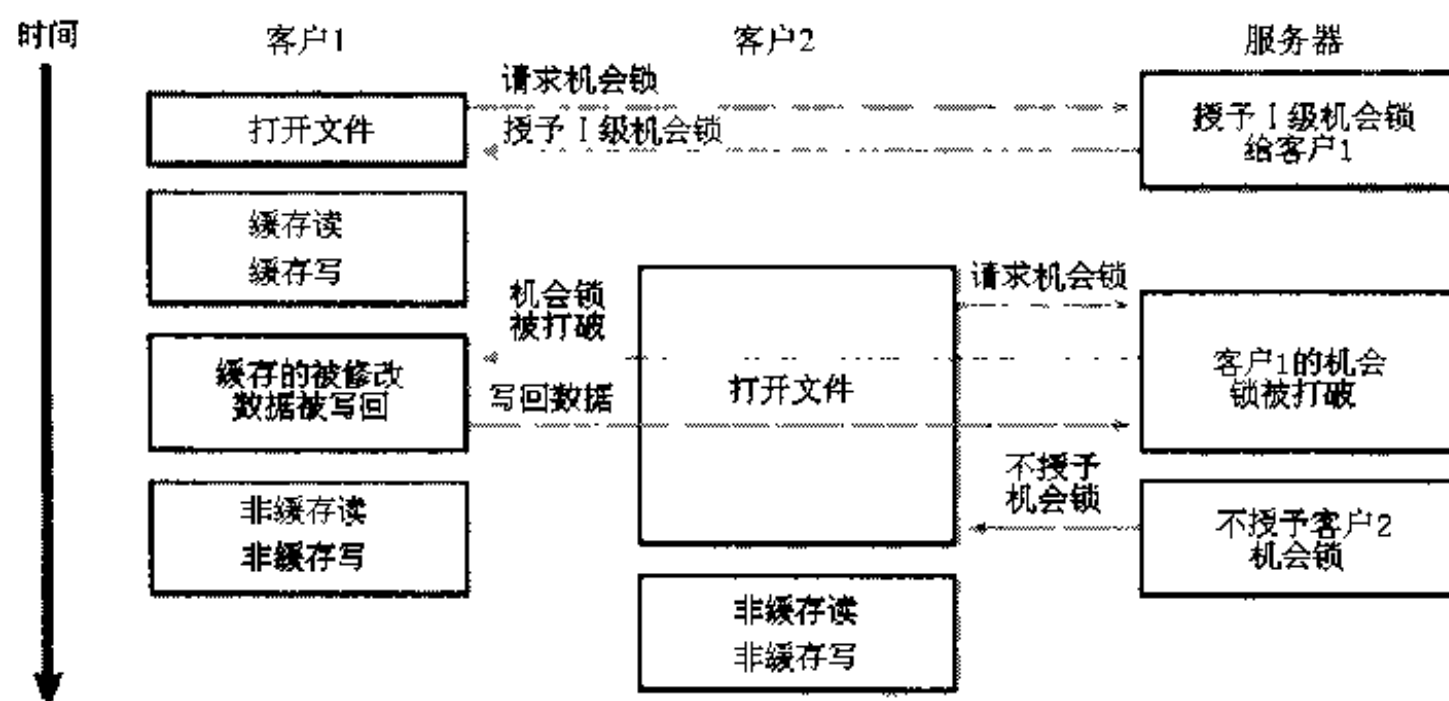


图7-12 一个机会锁的例子

5. NetBIOS

直到20世纪90年代，网络基本输入输出系统（NetBIOS）编程API是PC上使用最广的网络编程API。NetBIOS支持可靠的面向连接的通信和非可靠的无连接的通信。Windows 2000为了运行基于NetBIOS的遗留程序，提供对NetBIOS的支持。微软建议应用程序开发者不要再使用NetBIOS，因为其他API，比如命名管道和Winsock，与之相比更灵活更可移植。在Windows 2000上是通过TCP/IP、NetBEUI和IPX/SPX协议来支持NetBIOS的。

(1) NetBIOS名称

在NetBIOS的命名规范中，所有的计算机和网络服务器被分配了16字节长的名称，称为NetBIOS名称。NetBIOS名称有两种类型：UNIQUE和GROUP。对于UNIQUE类型的NetBIOS名称，网络中只能有一个实例；而对于GROUP类型的NetBIOS名称则可以有多多个网络程序拥有该名称。客户可以通过广播向一个GROUP的所有成员发消息。在Windows的网络服务器中，NetBIOS名称的第16个字节是保留的，表示该名称的资源类型。

为了支持与Windows NT4操作系统的互操作（还有Consumer Windows），Windows 2000为每一个域定义了一个NetBIOS名称。它是域的DNS名称的前15个字节。比如名为mspress.Microsoft.com的域，它的NetBIOS名称为mspress。同样，在安装每一台机器时Windows 2000需要管理员为它们分配相应的NetBIOS名称。

另一个NetBIOS中涉及的概念是LANA号。每块网络适配器上的每一个NetBIOS兼容协议都会分配一个LANA号。比如，一台机器有两块网卡，每块网卡都支持TCP/IP和NetBEUI协议，那么这台机器就有四个LANA号。LANA号十分重要，因为基于NetBIOS的应用程序必须显式地把服务器名称分配到每一个可接受相应用户连接的LANA号上。如果应用程序监听特定名称的用户连接，客户只能通过那些注册了该名称的LANA号，即相应网卡上的相应NetBIOS兼容协议来访

问应用程序。

一个名为Windows因特网名字服务 (Windows Internet Name Service, WINS) 的网络服务器维护NetBIOS名称和IP地址的映射。如果WINS没有安装, NetBIOS使用名称广播在Windows网络中传播NetBIOS名称。值得注意的是, NetBIOS名称排在DNS名称之后。计算机名称首先通过DNS注册并被解析, 只有当DNS域名解析失败之后, Windows 2000才进行NetBIOS名称的解析 (DNS域名解析将在本章的“网络资源名字解析”部分介绍)。

(2) NetBIOS的操作

NetBIOS服务器应用程序通过NetBIOS API枚举所在系统的所有LANA, 把表示应用程序的NetBIOS名称分配到每个LANA。如果服务器是面向连接的, 应用程序执行NetBIOS监听命令, 等待客户的连接请求。与客户建立连接后, 服务器应用程序执行NetBIOS命令进行数据的发送和接收。无连接的服务是类似的, 只不过服务器应用程序不用建立连接就可以接收消息。

面向连接的客户使用NetBIOS命令与服务器建立连接, 此后调用其他命令发送和接收数据。一个建立的NetBIOS连接是一个会话 (session)。如果客户想发送无连接的消息, 它只需在发送命令中指明服务器应用程序的NetBIOS名称就可以了。

NetBIOS由一系列命令构成, 不过这些命令通过一个单一的NetBIOS接口被调用。这种调用方案是由遗留程序造成的。在MS-DOS上NetBIOS是通过MS-DOS中断服务实现的。一个NetBIOS应用程序执行一个MS-DOS中断, 向MS-DOS的NetBIOS服务器发送一个数据结构。这个数据结构表示了将被执行的NetBIOS命令的所有细节。因此在Windows 2000下, Netbios函数只接受一个参数。它是一个数据结构, 包含了应用程序调用NetBIOS服务器的所有参数。

(3) NetBIOS API的实现

如图7-13所示, 是实现NetBIOS API的组件。Netbios函数通过\Winnt\System32\Netapi32.dll提供给应用程序。Netapi32.dll打开一个名为NetBIOS仿真驱动程序 (NetBIOS emulation driver) 的核心态驱动程序的句柄 (Winnt\System32\Drivers\Netbios.sys), 并代表应用程序发出Win32 DeviceIoControl命令。NetBIOS仿真驱动程序把收到的NetBIOS命令转接成TDI命令发送给相应的协议驱动程序。

如果一个应用程序使用基于TCP/IP的NetBIOS, 那么NetBIOS仿真驱动程序需要有NetBT驱动程序 (Winnt\System32\Drivers\Netbt.sys)。NetBT是基于TCP/IP的NetBIOS驱动程序, 它负责利用TCP/IP协议实现NetBIOS的语义。因为最初NetBIOS的语义是NetBEUI协议 (在本章的后面介绍) 固有的, 比如NetBIOS依赖NetBEUI的消息模式传输和NetBIOS名字解析, 所以这都需要NetBT驱动程序利用TCP/IP加以实现。同样, NwLinkNB驱动程序通过IPX/SPX协议实现NetBIOS的语义。

7.2.2 网络资源的名字解析

应用程序可以通过两种方法查询和访问远程机器上的资源。一种是使用UNC标准, 通过Win32函数直接访问远程资源。另一种方法是使用微软两网络 (WNet) API枚举所有计算机提供的可共享的计算机和资源。两种方法都使用重定向器访问网络上的资源。正如我们前面叙述过的, 客户要访问CIFS服务器是通过CIFS重定向器实现的, 而CIFS重定向器包括核心态的重定向器

FSD和用户态的Workstaion服务。微软也提供了访问Novell NetWare服务器的共享资源的重定向器，第三方也可以向Windows 2000添加自己的重定向器。在本章我们将研究当有远程I/O请求时，决定哪一个重定向器被调用的软件。它们是以下两个组件：

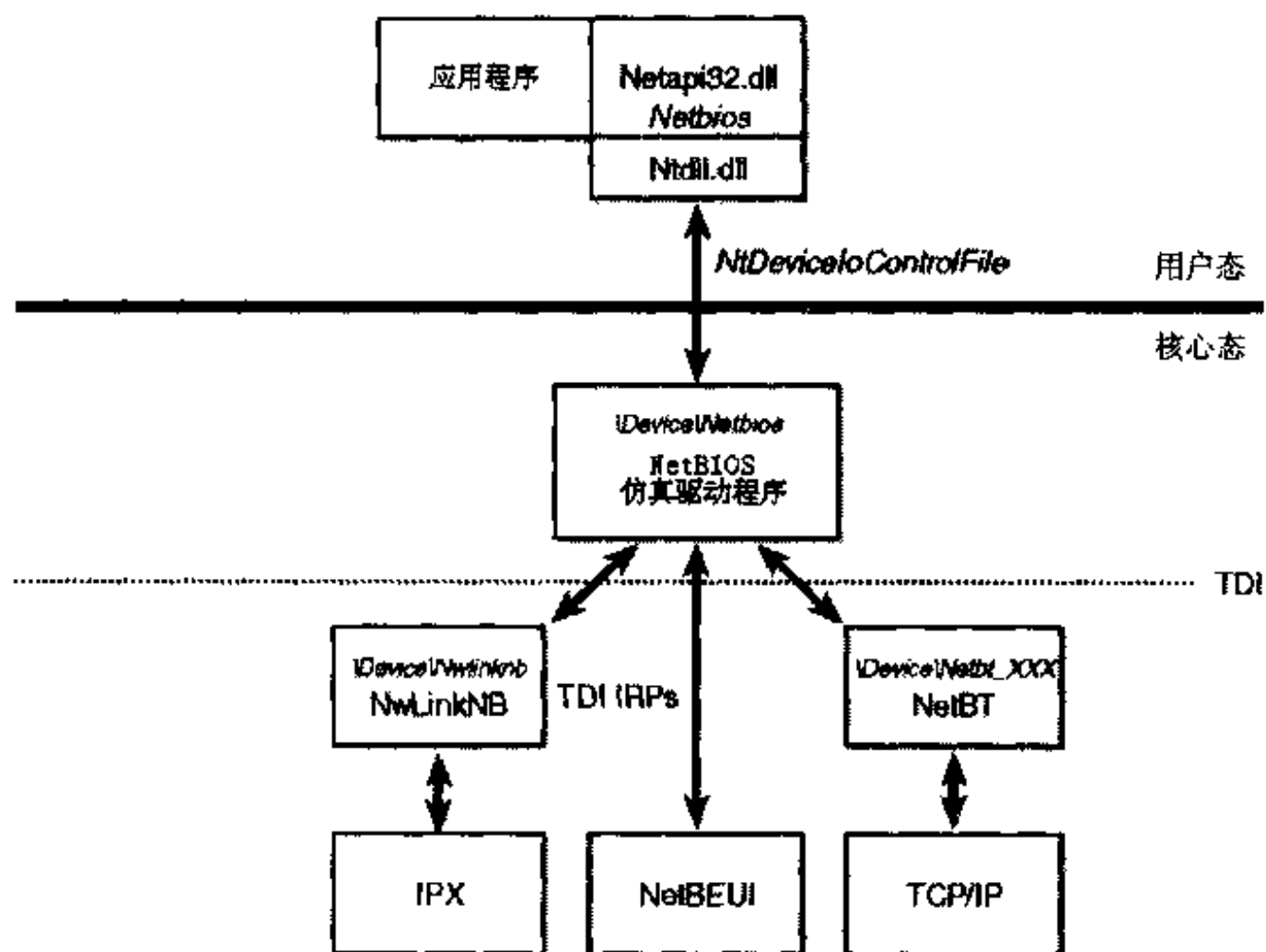


图7-13 NetBIOS API的实现

- 多提供者路由器（Multiple provider router, MPR）是一个DLL。它决定一个应用程序通过Win32 WNetAPI访问远程文件时使用何种网络。
- 多UNC提供者（Multiple UNC Provider, MUP）是一个驱动程序。它决定一个应用程序通过Win32 I/O API访问远程文件时使用何种网络。

最后我们将以Windows 2000的计算机名字解析系统的核心——域名系统（DNS）——结束这一节。

1. 多提供者路由器

Win32 WNet函数允许应用程序（包括Windows资源管理器、网上邻居应用程序）连接到各类网络资源，比如文件服务器和打印机，同时浏览远程文件系统上的各类内容。因为WNetAPI的调用可以跨越各类网络，使用各类网络传输协议，系统必须使请求正确地经网络发送并理解远程文件系统发回的结果。图7-14描述了重定向器如何负责这一工作。

提供者（provider）是这样的一类软件，它们使Windows 2000作为一个客户连接到远程网络服务器上。WNet提供者完成以下一些操作，包括建立和断开网络连接，远程打印，数据传输。内置的WNet提供者包括一个DLL、Workstation服务和重定向器。其他网络供应商只需要提供DLL和重定向器。

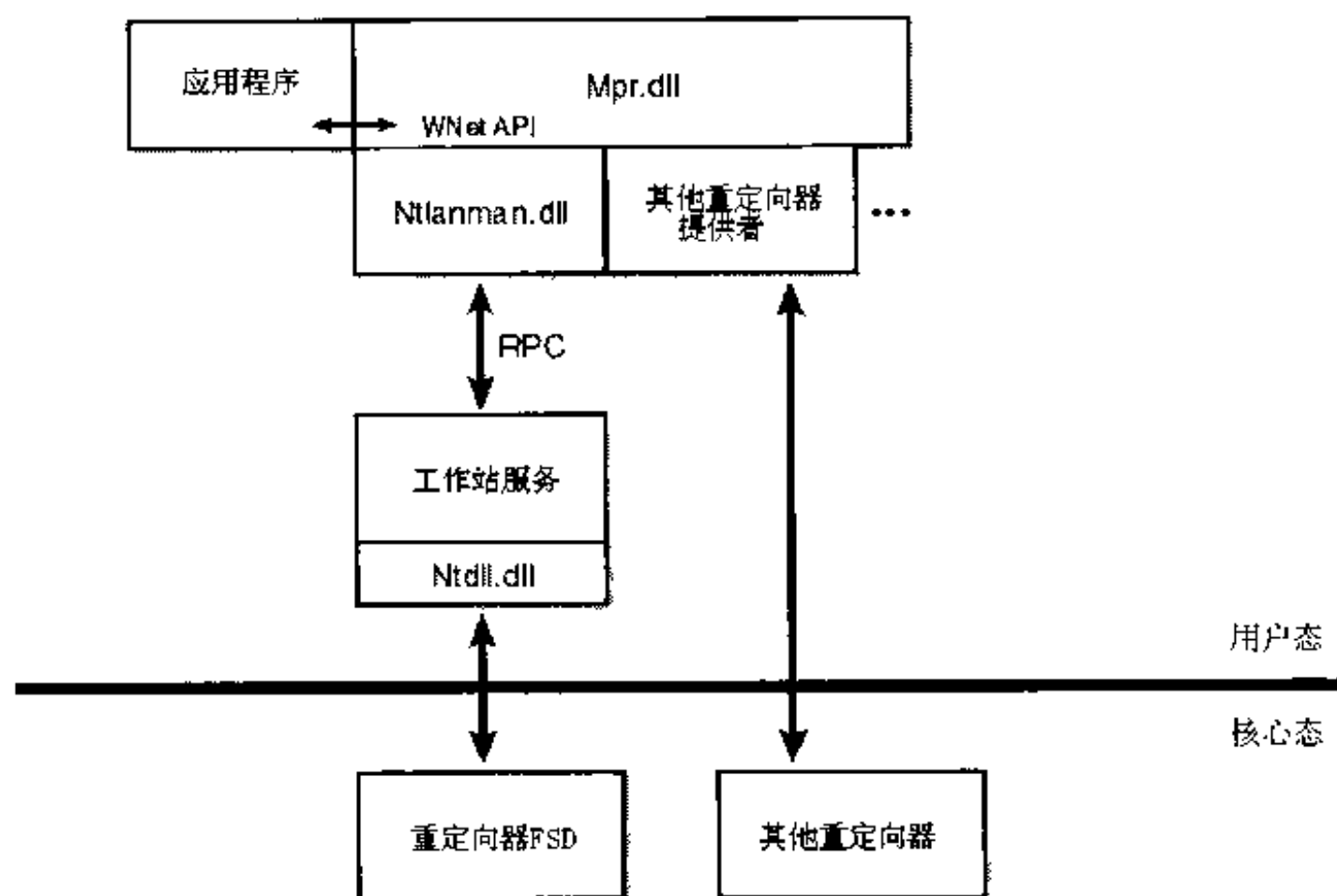


图7-14 MPR组件

当应用程序调用WNet例程时，调用被直接发送给MPR DLL。MPR接受调用，确定哪一个提供者可以识别要被访问的资源。MPR下的每一个提供者DLL都提供了被称为提供者接口（provider interface）的一套标准函数集。这个接口可以使MPR确定应用程序将要访问哪一个网络并把调用请求发给合适的WNet提供者。重定向器FSD的提供者是\Winnt\System32\Ntlanman.dll，这是由注册表HKLM\SYSTEM\CurrentControlSet\Services\lanmanworkstation\NetworkProvider主键ProviderPath的键值确定的。

当被调用WnetAddConnection API函数连接一个远程网络资源的时候，MPR检查HKLM\SYSTEM\CurrentControlSet\Control\Network Provider\Order\ProviderOrder键值，确定加载哪一个网络提供者。MPR按照它们在注册表中的次序轮询每一个提供者，直到有一个相应的重定向器识别这一资源或者所有可用的提供者都被选取。可以通过高级设置对话框改变ProviderOrder，如图7-15所示。（图上所示的系统只有一个提供者）这个对话框可以通过选取网络和拨号连接应用程序的高级菜单来访问。可以右键点击桌面上网上邻居图标，选择弹出式菜单的属性或者直接点击

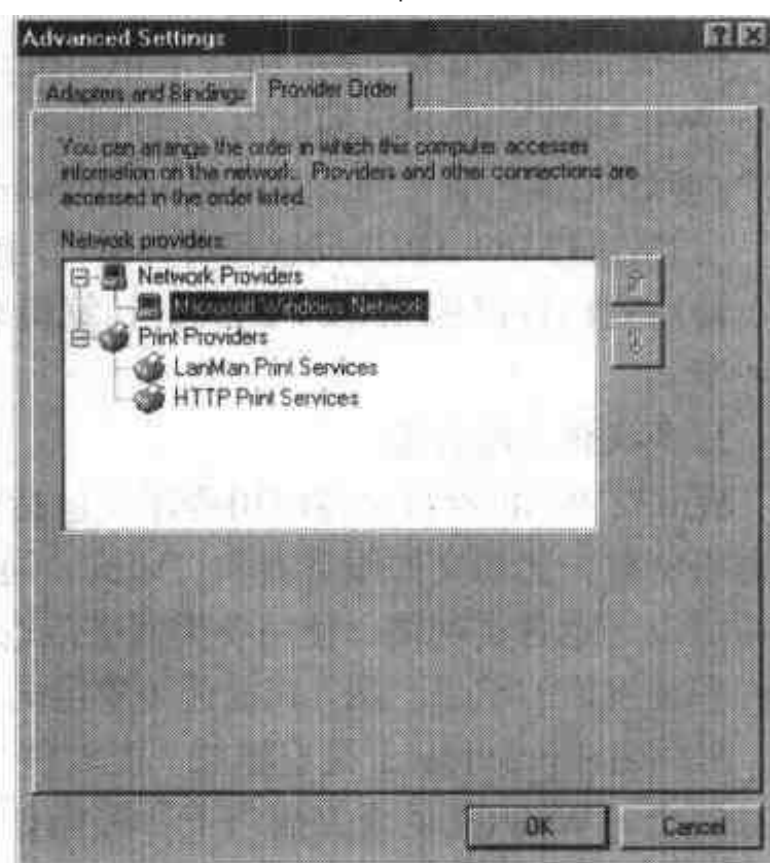


图7-15 提供者次序编辑器

开始菜单的设置选项来选取网络和拨号连接应用程序。

WNetAddConnection函数为每个远程资源分配一个驱动器号或者设备名称。当该函数被调用时，它把调用传给合适的网络提供者。相应地，网络提供者在对象管理器名字空间中建立一个符号连接对象。这个对象把定义的驱动器号映射到相应网络的重定向器（通常是远程FSD）。

图7-16显示了\??目录，你可以看到代表远程文件共享连接的几个驱动器号。图上显示重定向器创建了名为\Device\LanmanRedirector的设备对象。符号连接中另外的文本告诉重定向器驱动器号对应的远程资源是什么。如果用户打开文件X:\Book\Chap13.doc，重定向器将接收到通过符号连接解析后剩下的路径的未解析部分，即“;X:0\duall\Book\Chap13.doc”。由此重定向器知道要访问的资源在dual服务器的共享驱动器E上。

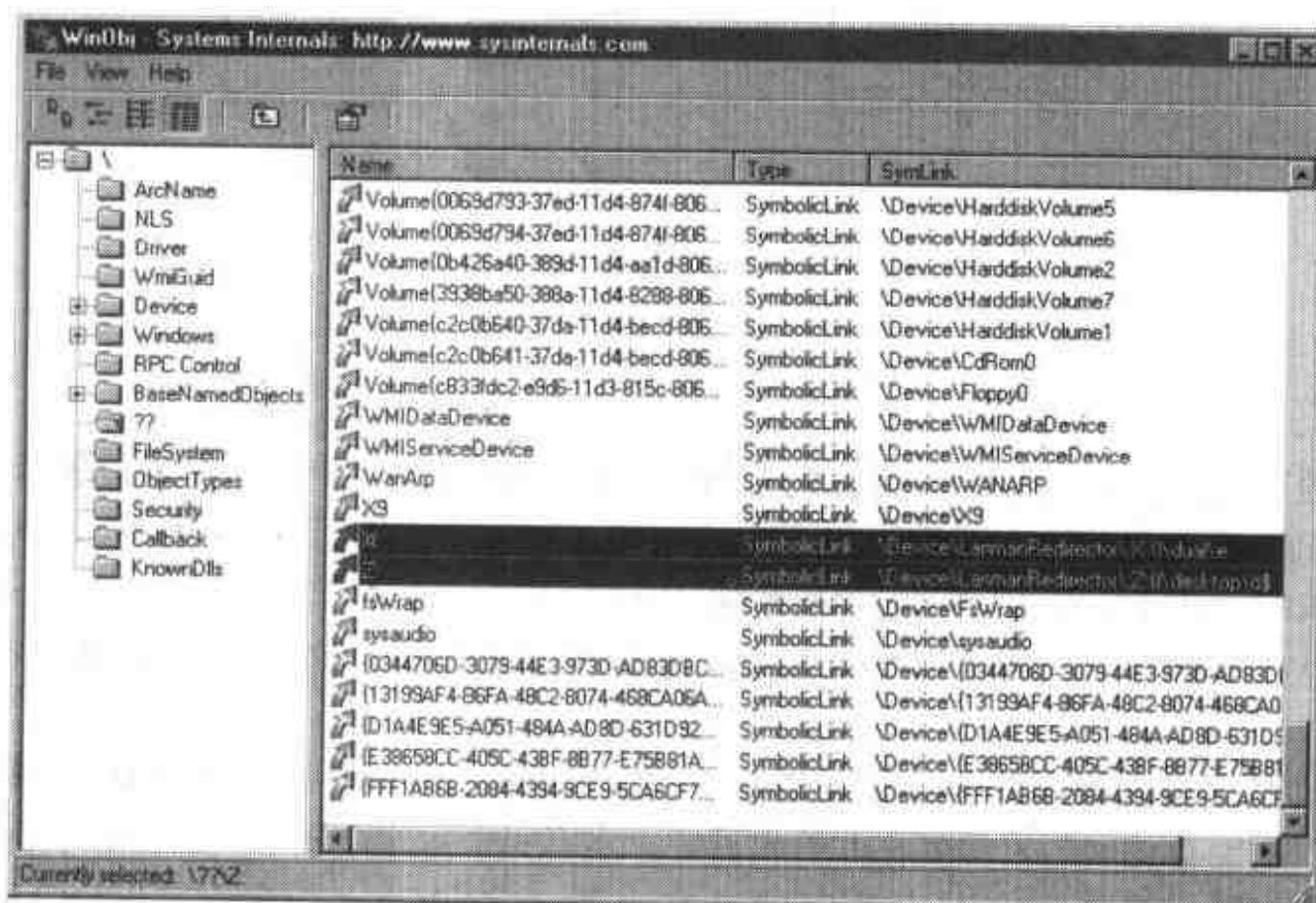


图7-16 解析网络资源名称

和内置的重定向器一样，其他重定向器在被载入系统并被初始化时也在对象管理器名字空间创建设备对象。当WNet或者其他API调用对象管理器打开不同网络上的资源时，对象管理器使用设备对象作为访问远程文件系统的出发点。它调用设备对象相应的I/O管理器的分析方法，确定可以处理请求的重定向器FSD。

2. 多UNC提供者

多UNC提供者（MUP）是一个和MPR类似的网络组件。所有针对UNC名称的文件或设备的I/O请求都由它处理（以\\开头的名称表明相应的资源存在于网上）。像MPR一样，MUP接收请求，确定哪一个本地重定向器可以识别远程资源之一。与MPR不同的是，MUP是一个设备驱动程序（在系统启动时被载入）。因此它直接向下层的驱动程序发出I/O请求。图7-17所示的例子中，它直接向重定向器发请求。

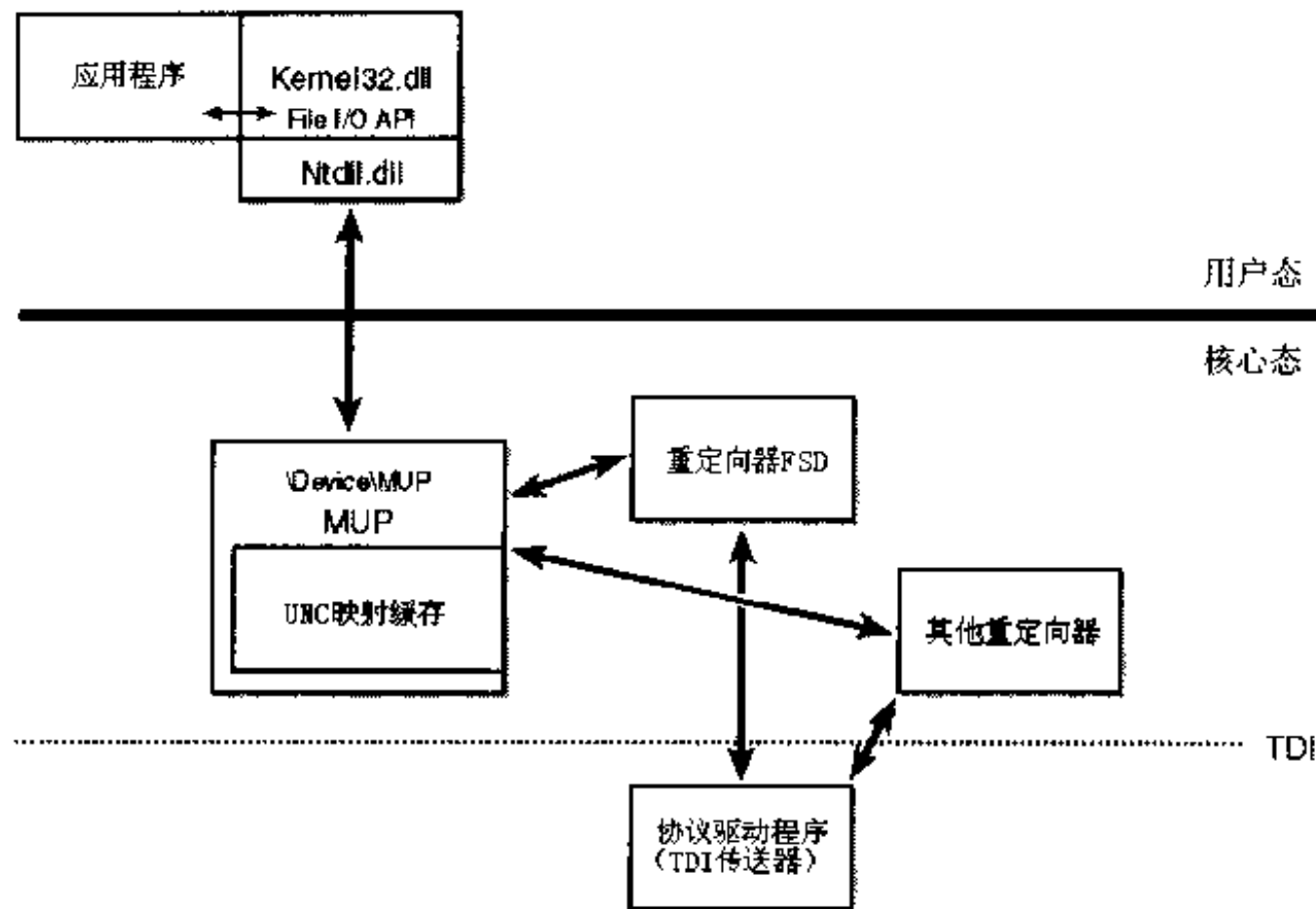


图7-17 多UNC提供者

当应用程序企图打开一个UNC名称的远程文件或设备时，MUP驱动程序被激活（UNC名称没有重定向驱动器号）。当Win32客户DLL Kernel32.dll（一个提供了文件I/O方面API的DLL）接收到调用请求时，该子系统在UNC名称前添加“\\?\UNC”的前缀字符串并调用NtCreateFile打开这一文件。“\\?\UNC”作为符号连接名称被解析成\\Device\Mup，这是一个表示MUP驱动程序的设备名称。

MUP驱动程序接受调用请求，此后向每一个注册的重定向器发出一个异步的IRP，等待它们中的一个能识别要被访问的资源名称并给出回应。当有一个重定向器识别出资源名称后，它将指出这个名称的哪一部分是它可以确定的。举个例子来说，比如名为\\WIN2KSERVER\PUBLIC\insidew2k\chap13.doc的文件，重定向器识别出它以后会声明字符串\\WIN2KSERVER\PUBLIC是属于自己的。MUP驱动程序将缓存这一信息，此后所有以那个字符串开头的请求将直接发往那个重定向器，从而省去了轮询过程。MUP驱动程序的缓存有超时失效的特征，如果一段时间没被用到，与一个重定向器相关联的字符串将会失效。如果多个重定向器识别出同一资源，MUP驱动程序将根据注册表中ProviderOrder列出的重定向器的次序决定哪个重定向器优先得到这个I/O请求。

3. 域名系统

域名系统（DNS）是因特网名称转换成IP地址的标准。一个网络应用程序如果想把DNS名字解析成IP地址，必须通过TCP/IP协议向DNS服务器发送一个DNS查找请求。DNS服务器使用分布式数据库存储‘名称/IP对’进行名称到IP地址的翻译。每一个服务器负责一定区域（zone）的翻译。有关DNS的详细描述超出了本书的范围，但DNS是Windows 2000中命名的基础，它是

Windows 2000最主要的名称解析协议。

Windows 2000 DNS服务器作为一个Win32服务（\Winnt\System32\Dns.exe）包括在Windows 2000的服务器版本中。标准的DNS服务器实现使用文本文件作为翻译数据库，但Windows 2000 DNS服务器可以改用活动目录存储相应区域的信息。

7.2.3 协议驱动程序

网络API驱动程序接受API请求，把它们转换为底层网络协议的传输请求。API驱动程序依赖核心态的传输协议驱动程序进行实际的转换。API和下层的网络协议是分开的，使得整个网络体系结构十分灵活，它允许每个API使用不同的网络协议。Windows 2000带有DLC、NetBEUI、TCP/IP和NWLink传输协议的驱动程序。另外一些协议的驱动程序是可选的，比如Windows 2000服务器版本安装Services For Macintosh时安装的AppleTalk协议。这里对每一个协议作简单的介绍：

- DLC协议是一种相对原始的协议。IBM的一些大型机和HP的一些网络打印机使用了这一协议。它是一种“原始”协议。因为没有一种网络API可以使用它。想要使用DLC的应用程序必须直接调用DLC传输协议设备驱动程序的接口。
- IBM和微软在1985年引入了NetBEUI，微软把NetBEUI作为LAN管理器和NetBIOS API的默认协议。微软此后增强了NetBEUI，但这个协议有局限性，它是非路由的，同时在WAN上性能很差。NetBIOS扩展用户接口（NetBIOS Extended User Interface, NetBEUI），之所以这样命名是因为它和NetBIOS紧密集成在一起。微软NetBEUI协议驱动程序实现的NetBEUI采用NetBIOS帧（NBF）格式。Windows 2000支持NetBEUI只是为了和以前的Windows操作系统（Windows NT 4和Consumer Windows）互操作。
- 因特网的飞速发展和对TCP/IP协议的依赖使得TCP/IP协议成为Windows 2000最主要的协议。美国国防部高级研究计划局（DARPA）在1969年开发的ARPANET是因特网的基础。TCP/IP有适于WAN的特征，比如可路由和在WAN上较好的性能。TCP/IP协议是Windows 2000优先使用的协议，也是唯一缺省安装的协议。
- NWLink协议是由Novell的IPX协议和SPX协议组成的。Windows 2000中包括NWLink协议是为了和Novell NetWare服务器互操作。

Windows 2000的传输口通常实现了上述传输协议的所有相关协议。比如TCP/IP驱动程序（\Winnt\System32\Drivers\Tcpip.sys）支持TCP、UDP、IP、ARP、ICMP和IGMP。一个TDI传输口通常使用设备对象表示特定的协议。客户获取表示协议的文件对象，使用IRP向对应协议发出网络I/O请求。TCP/IP驱动程序创建三个设备对象，表示TDI客户会使用的三种协议：\Device\TCP、\Device\Udp和\Device\Ip。

网络API不必对每一种传输协议使用不同的接口，微软制订了传输驱动程序接口（TDI）标准。正如本章前面提到的，TDI接口实质上是网络请求如何格式化成IRP以及网络地址和通信连接如何分配的规范。支持TDI规范的传输协议向AFD和重定向器之类的客户提供接口。通过Windows 2000设备驱动程序实现的传输协议被称为TDI传送器。因为TDI传送器是设备驱动程序，它们只处理客户发来的IRP格式请求。

在\Winnt\System32\Drivers\Tdi.sys库中的支持函数和开发者在自己驱动程序中的定义构成了TDI接口。TDI编程模型和Winsock十分相似。一个TDI客户执行以下步骤和远程服务器建立连接：

1) 客户分配并格式化一个address open TDI IRP来请求分配一个地址。TDI传送器返回一个被称为地址对象（address object）的文件对象。这一步等价于Winsock的bind函数。

2) 客户分配并格式化一个connection open TDI IRP。TDI传送器返回一个被称为连接对象的文件对象来表示连接。这一步等价于Winsock的socket函数。

3) 客户用associate address TDI IRP把连接对象和地址对象关联在一起（在Winsock中没有等价的函数）。

4) 接收远程连接的TDI客户发出listen TDI IRP，该IRP指明了服务器端连接对象能支持的最大连接数。同时该TDI客户发出accept IRP。当建立远程连接（或者出现错误）时，TDI传送器完成accept IRP。这一步等价于Winsock的listen和accept函数。

5) 想和远程服务器建立连接的TDI客户发出connect IRP，该IRP指明了连接对象。当建立远程连接（或者出现错误）时，TDI传送器完成connect IRP。这一步等价于Winsock的connect函数。

TDI也支持UDP等无连接协议使用无连接通信。此外TDI还支持这样的方式：TDI的客户在TDI传送器中注册一个事件回调，当TDI传送器从网络上获得数据后，TDI传送器直接调用事件回调函数。TDI这种基于事件的回调机制使得TDI传送器能够通知它的客户相关的网络事件。使用事件回调的客户在接受网络数据时不必预先分配缓冲等资源，因为它们可以访问TDI协议驱动程序缓冲中的数据。

7.2.4 NDIS驱动程序

当一个协议驱动程序在网络上读写相应协议格式的消息时，必然要使用网络适配器。不可能指望协议驱动程序了解市场上每一块网络适配器的细微差别（有专利的网络适配器就有几千种）。只有靠网络适配器供应商提供驱动程序，利用驱动程序在他们的硬件上传输和接收网络消息。1989年，微软和3COM联合开发了网络驱动程序接口规范（NDIS），它允许协议驱动程序以与设备无关的方式和网络适配器驱动程序通信。遵守NDIS的网络适配器驱动程序被称为NDIS驱动程序或NDIS小端口驱动程序。Windows 2000中的NDIS版本是NDIS 5。

在Windows 2000中，NDIS库（\Winnt\System32\Drivers\Ndis.sys）实现了TDI传送器和NDIS驱动程序的边界。和Tdi.sys一样，NDIS库是一个辅助库，NDIS驱动程序的客户使用这个库格式化发向NDIS驱动程序的命令。NDIS驱动程序通过NDIS库接收请求或者发回响应。图7-18显示了NDIS相关各组件的关系。

微软的网络体系结构的设计目标之一就是使适配器供应商能够很容易地开发NDIS驱动程序并很容易地在Consumer Windows和Windows 2000之间进行移植。除了提供NDIS边界辅助例程，NDIS库为NDIS驱动程序提供了完整的执行环境。如果没有NDIS库对它们的封装，它们将不能被调用。因此NDIS驱动程序不是真正的Windows 2000驱动程序。NDIS驱动程序被隔离层完全地封装，不能直接接收和处理IRP。NDIS库从TDI服务器接收IRP，把它们转换成对NDIS驱动程序的调用。在驱动程序完成上次调用之前，NDIS库可能向驱动程序发出新的调用，NDIS库保证NDIS

驱动程序不必担心这种可重入性问题。没有可重入性问题意味着在编写NDIS驱动程序时不必考虑复杂的同步问题。在多处理器并行处理的环境下，这可能是一个非常棘手的问题。

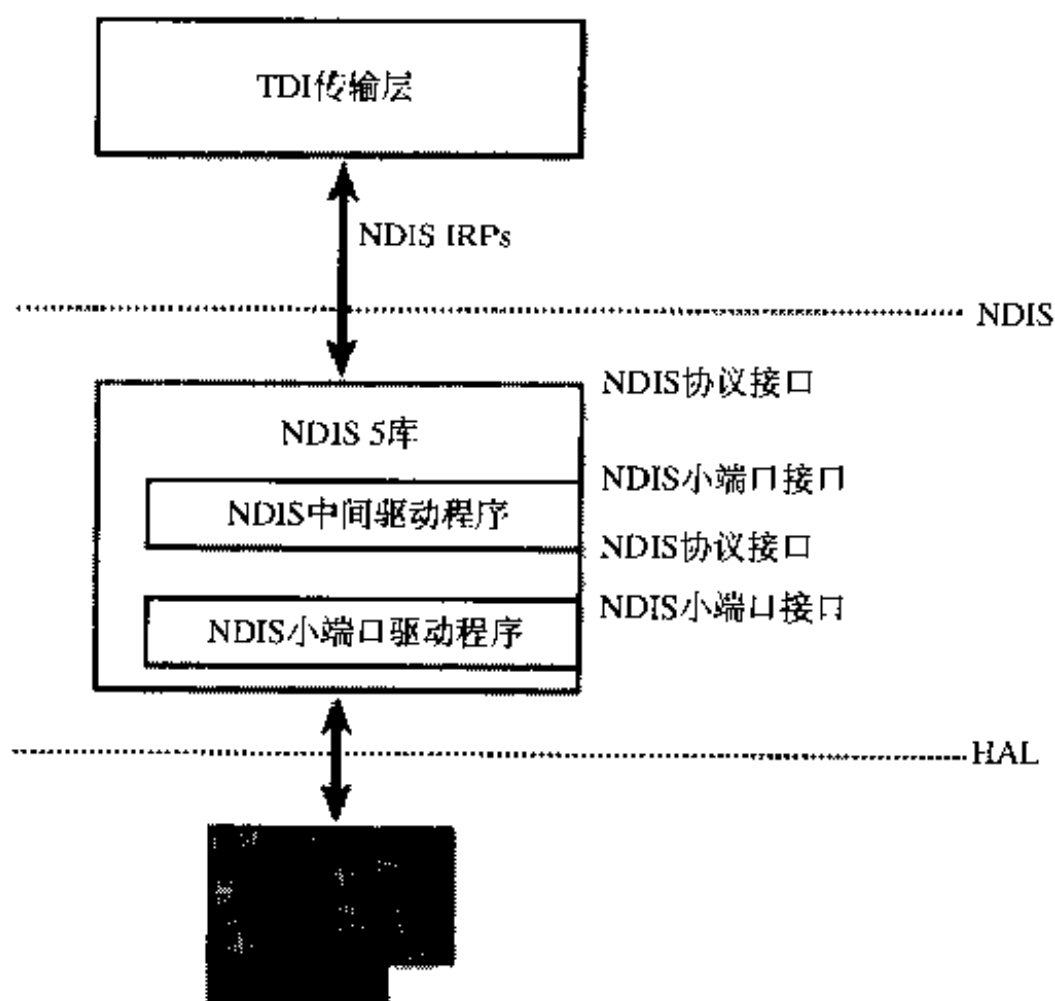


图7-18 NDIS的组件图

尽管NDIS库对NDIS驱动程序的序列化简化了开发，但序列化会影响多处理器可扩展性。在多处理器环境，标准的NDIS 4驱动程序（NDIS库的Windows NT4版本）对某些操作的可扩展性不是很好。在NDIS 5中，微软允许开发者对操作反序列化。NDIS 5驱动程序可以向NDIS库表明它们不想被序列化，那么NDIS库将把收到的IRP请求直接发给驱动程序。这时NDIS驱动程序必须负责对请求的排队和并发管理。在多处理器环境中，这种反序列化可获得更好的性能。

1. NDIS小端口的变体

NDIS模型也支持被称为NDIS中间驱动程序（NDIS intermediate driver）的混合TDI传送器-NDIS驱动程序。这类驱动程序处于TDI传送器和NDIS驱动程序之间。对于NDIS驱动程序，NDIS中间驱动程序看上去像一个TDI传送器，对于TDI传送器，NDIS中间驱动程序看上去像一个NDIS驱动程序。因为NDIS中间驱动程序处在协议驱动程序和网络驱动程序中间，它可以处理系统所有网络通信的数据分组。支持网络适配器故障容错和负载均衡的软件，比如微软网络负载均衡提供者，就要依靠NDIS中间驱动程序。QoS中的分组调度器也是NDIS中间驱动器的一个例子。

2. 面向连接的NDIS

NDIS 5引入了新的一类NDIS驱动程序——面向连接的NDIS小端口驱动程序。由此Windows 2000天生具备了对面向连接的网络硬件（比如ATM）的支持，Windows 2000网络体系结构对连接的管理和建立也有了标准。面向连接的NDIS使用许多和标准NDIS相同的API，但面向连接的NDIS

驱动程序通过已建立的网络连接发送数据分组，而不只是简单地把数据分组放到网络媒介上。

除了支持而向连接的网络媒介的NDIS小端口驱动程序，NDIS 5也定义了一些驱动程序支持面向连接的小端口驱动程序。

- 通话管理器是一种NDIS驱动程序，向面向连接的客户id提供通话建立和挂断服务。通话管理器使用面向连接的小端口驱动程序和其他网络实体，比如网络交换机或其他通话管理器，交换信号信息。一个通话管理器支持一个或多个信号协议，比如ATM用户网络接口3.1 (UNI)。
- 一个整合的小端口通话管理器 (MCM) 是一个而向连接的小端口驱动程序。它向面向连接的客户id提供了通话管理器服务。MCM实质上是内置通话管理器的NDIS小端口驱动程序。
- 一个而向连接的客户使用通话管理器或者MCM的通话建立和挂断服务，使用面向连接的NDIS小端口驱动程序的发送和接收服务。一个面向连接的客户可以向网络协议栈的高层提供自己协议的服务，或者它可在无连接协议和面向连接的介质之间充当仿真层。LAN仿真 (LANE) 就是一个例子，它向上层协议隐藏了ATM而向连接的特征，表现为一种无连接的媒介。

图7-19显示了这些组件之间的关系。

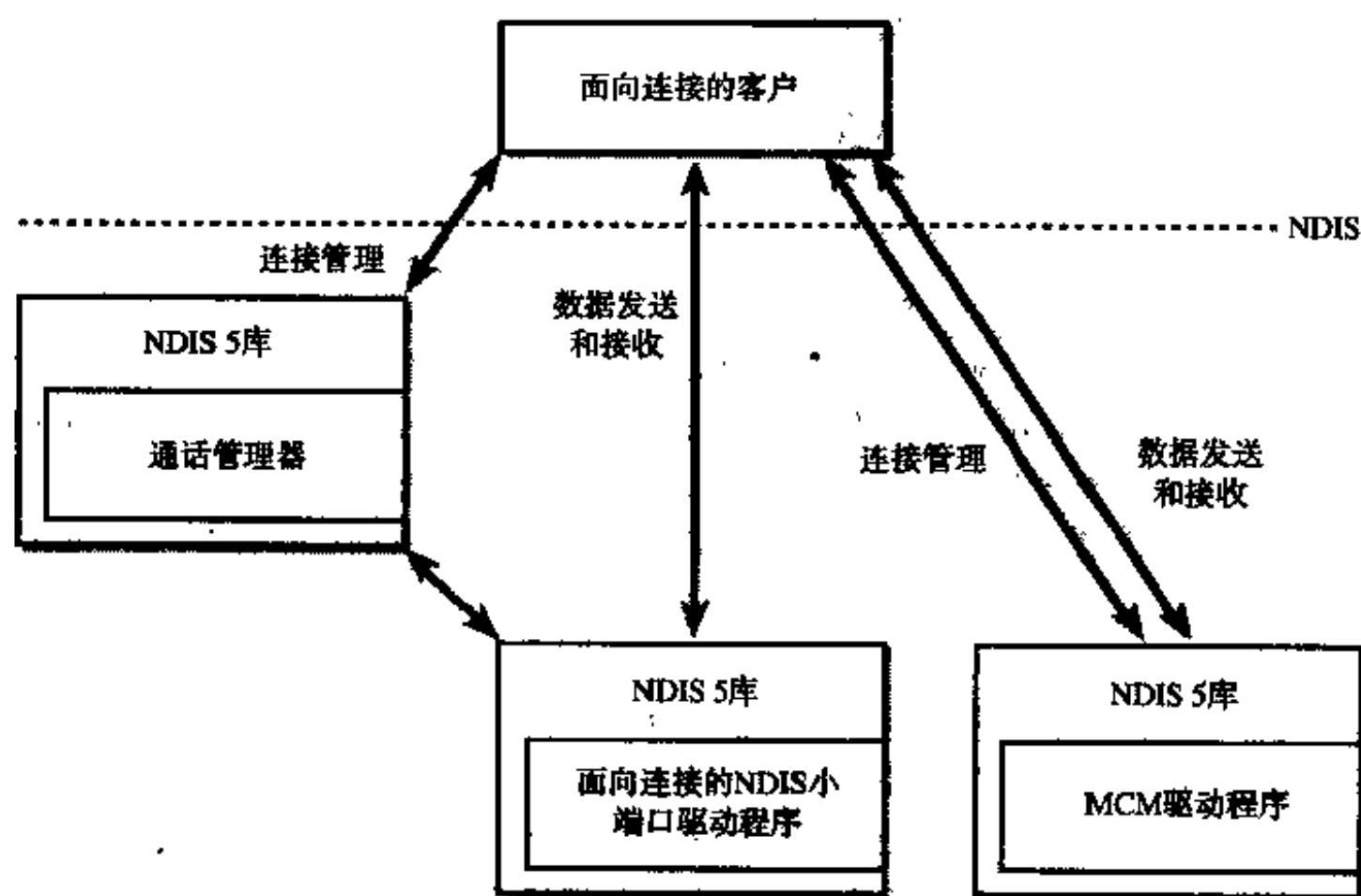


图7-19 面向连接的NDIS驱动程序

7.3 Windows 2000的层次化网络服务

本章中我们讨论了Windows 2000中建立在API及组件之上的网络服务。尽管对这些服务的功能以及内部具体实现的说明已超出了本书的范围，但是本节将简略介绍远程访问、活动目录、网

络负载均衡、文件复制服务（FRS）以及分布式文件系统（DFS）。另外，Windows 2000支持几种基于TCP/IP协议的扩展特性的服务，它们包括网络地址翻译（NAT）、网际协议安全性（IPSec）以及服务质量（QoS）。

7.3.1 远程访问

Windows 2000支持远程访问，它允许远程访问的客户连接远程访问服务器并访问网络资源，例如文件、打印机以及网络服务。这样，客户就好像与远程访问服务器的网络连在了一起。Windows 2000提供两种远程访问类型：

- 客户使用拨号远程访问经过电话线或其他电信设施接入远程访问服务器。电信介质用于建立一条客户与服务器之间的临时物理或虚拟连接。
- 虚拟专用网络（VPN）的远程访问使VPN客户与服务器建立一条点到点的虚拟连接，这种连接一般建立在IP网络上，如因特网。

远程访问与远程控制不同，因为远程访问起到代理连接Windows 2000网络的作用，而远程控制软件必须在服务器上运行应用程序，用于给客户提供一个用户接口。

7.3.2 活动目录

活动目录是轻量目录访问协议（LDAP）中目录服务的Windows 2000的实现方式。活动目录基于存储资源对象的数据库，这些对象由Windows 2000网络中的应用程序定义。例如Windows 2000域的成员以及结构，包括用户账号和密码信息都存储在活动目录中。

我们称对象类及其属性为模式，活动目录模式中的对象以层次化布局。就像注册表内的逻辑组织一样，其中容器对象能够储存其他对象。

活动目录支持大量的API，它们能让客户在活动目录的数据内访问对象：

- LDAP C API是C语言版本的API，它使用LDAP网络协议。C或C++应用程序可以直接使用此API，其他语言编写的应用程序需要通过翻译层访问这些API。
- 活动目录服务接口（ADSI）是活动目录的COM接口，它屏蔽了LDAP的编程细节。ADSI支持多种语言，如Microsoft Visual Basic，C以及Microsoft Visual C++。ADSI也可以由Windows脚本主机（WSH）应用程序使用。
- 消息API（MAPI）兼容Microsoft Exchange客户软件和Outlook Address Book客户应用程序。
- 安全账号管理器（SAM）API建立在活动目录的最上层，它提供登录验证分组的接口，如MSV1_0（\Winnt\System32\Msv1_0.dll，它沿用以前NT LAN管理器的验证）和Kerberos（\Winnt\System32\Kdsc.dll）。
- Windows NT4网络API（Net API）用于NT4的客户通过SAM验证获取访问活动目录的权利。

活动目录是作为一个数据库文件实现的，文件名称是\Winnt\Ntds\Ntds.dit，此文件在域中的域控制器上复制。活动目录的目录服务是在局部安全授权子系统（Lsass）进程中执行的Win32服务。它管理数据库，使用实现数据库的磁盘结构（on-disk）的DLL，除此之外还提供用于保护数据库完整性的、基于事务的更新。活动目录数据库的存储还基于可扩展的存储引擎（ESE）数据

库，这种技术用于Microsoft Exchange Server 5.5版本的客户/服务器消息传送与群件系统。图7-20描述了活动目录的体系结构。

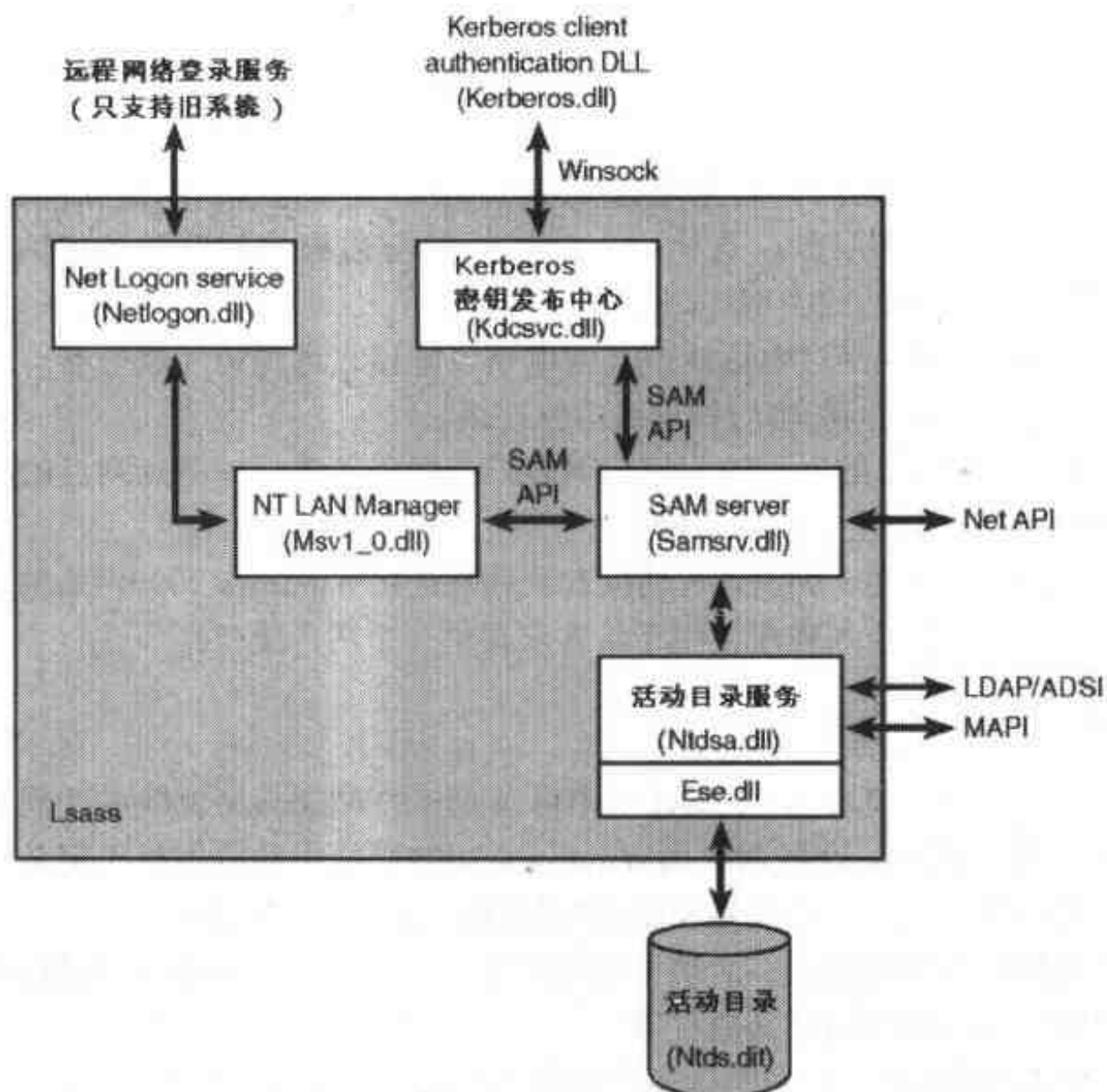


图7-20 活动目录的体系结构

7.3.3 网络负载均衡

正如本章之前所陈述的，Windows 2000 Advanced Server中的网络负载均衡服务是基于NDIS中间驱动程序。网络负载均衡允许建立一个可以多达32台计算机的集群，这些计算机在网络负载均衡服务中称为集群主机。集群服务器维护一个虚拟IP地址，并公开给客户访问，客户的请求能够送至集群中的所有计算机。然而，只有一台集群主机能够响应请求。网络负载均衡的NDIS驱动程序能够在有效的集群主机中，分布式地、高效地分配客户空间。这样，每台主机能够处理一部分的客户请求，每个客户请求总能被一台且只有一台主机所应答。集群主机一旦决定处理某个客户请求，它就将此请求送给TCP/IP协议驱动程序，最终到达服务器应用程序；而其他集群主机什么也不用做。如果一台集群主机不再运行，则集群中的其余主机得知那台主机不再参与处理请求后，就会将那台主机所获得的请求再发送给其余的主机。另外，任何一台集群主机可以添加到一个集群中去替换一台已有的主机，并且它能够做到无缝地开始处理客户请求。

网络负载均衡并不是一种集群技术的通用解决方案，因为与客户交互的服务器应用程序必须拥有某些特性：首先，它必须基于TCP/IP；其次，它必须能够在网络负载均衡集群内的任何一台系统中处理客户请求。第二个要求的含义是一个应用程序，它必须能够得知共享状态，这样就能为客户请求提供服务，应用程序本身必须能够管理共享状态——网络负载均衡服务并不包括自动的在集群主机中发布共享状态的服务。网络负载均衡中理想化的应用程序应包含用于提供静态内容的网站服务器、Windows媒体服务器以及终端服务。

7.3.4 文件复制服务

Windows 2000 Server包含文件复制服务（FRS）。复制域控制器\SYSTEMVOLUME_INFORMATION目录的内容是它的主要功能，\SYSTEMVOLUME_INFORMATION目录是Windows 2000域控制器保存登录脚本和组策略的地方（组策略允许管理员为域中的计算机定义使用及安全策略）。另外，FRS可以用于复制系统之间的分布式文件系统（DFS）的共享资源。FRS还提供分布式多主控复制，它能使任何一台服务器执行复制操作。复制的目录或文件一旦改变，这些变化会广播到其他域控制器上。

FRS中的基本概念是复制集，它由两个或两个以上的系统组成，这些系统能够根据管理制定的调度与拓扑，相互复制目录树中的内容。只有NTFS卷上的目录可以进行复制，因为FRS依赖于NTFS的更改历程，它需要侦测目录中的文件与复制集中的变化。因为FRS是基于多主控复制，它能够理论上支持成百上千台系统，并把它们作为复制集的一部分，复制集中的计算机能够在任意的网络拓扑中连接（例如环状、星型或者网状）。计算机也可以是多个复制集中的成员。

FRS是Win32的一个服务（\Winnt\System32\Ntfrs.exe），它使用经过验证的并附带通信加密的RPC。另外，因为活动目录包含它自身的复制能力，FRS使用活动目录API检索来自活动目录的FRS配置信息。

7.3.5 分布式文件系统

分布式文件系统（DFS）服务器位于工作站服务器的顶层，它将文件共享共同连入一个单一的名字空间。文件能够在同一台或者多台不同的计算机上共享，而且DFS可以为客户提供位置透明的资源访问。DFS名字空间的基础是一个文件共享必须在Windows 2000 Server中定义。

除了具有统一的网络资源名字空间的功能以外，DFS还提供DFS复制集的特性，它基于FRS复制集。管理员能够从两个或两个以上的共享中创建DFS复制集，这样FRS就能够在复制集中的共享之间复制数据，并保持其内容同步。通过随机选择复制集的成员来完成客户对复制集的数据请求，DFS提供负载均衡的有限形式。另外，当一个成员失效时，DFS利用对工作成员或复制集中成员的路由请求获得高可靠性。

DFS架构中的组件如图7-21所示。DFS的服务器端实现由一个Win32服务（\Winnt\System32\Dfsrv.exe）和一个设备驱动程序（\Winnt\System32\Drivers\Dfs.sys）组成。DFS服务器在注册表（非活动目录系统）或者在活动目录中，负责输出DFS拓扑管理接口以及维护DFS拓扑。当DFS收到客户请求时，驱动程序进行拓扑查询，然后将客户定向到请求的文件所在的系统。

在客户端，DFS的实现依赖于MUP驱动程序和NetWare与CIFS重定向器的支持。当客户发出

访问DFS名字空间中的文件I/O请求时，通过使用合适的重定向器，客户的MUP驱动程序与DFS服务器便可以进行通信。

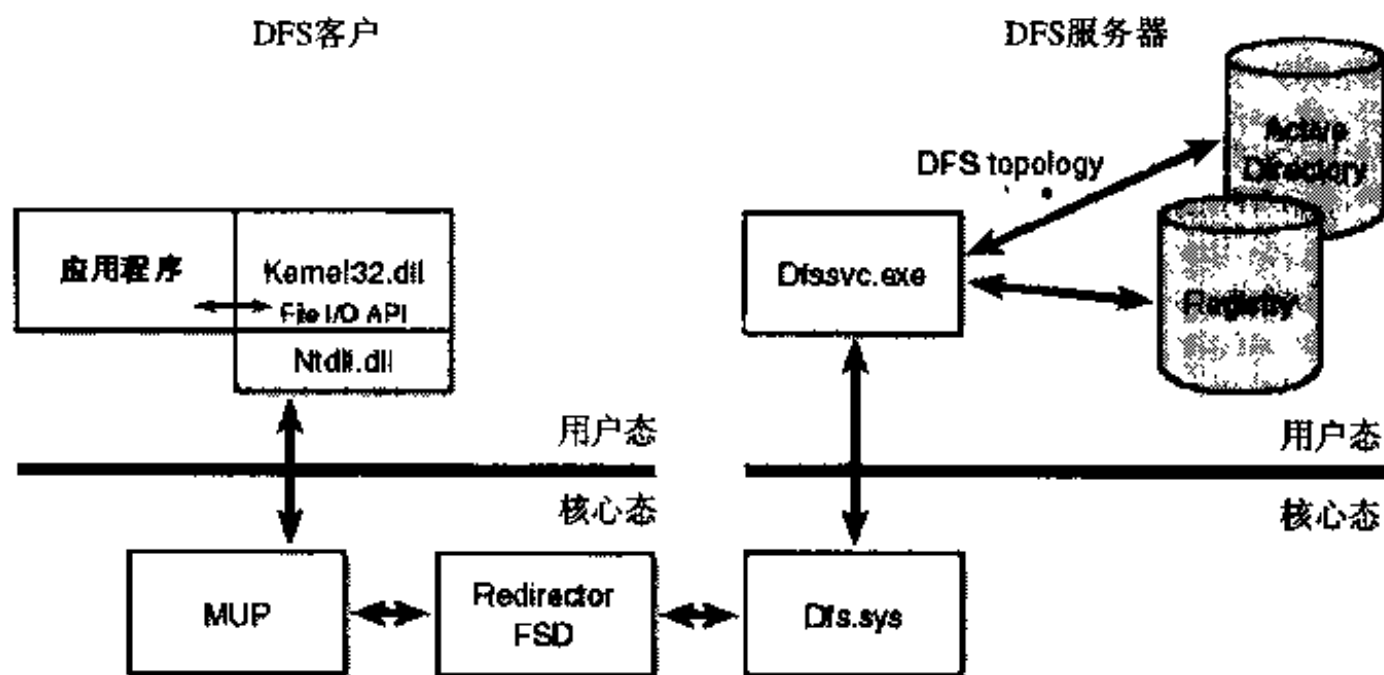


图7-21 DFS的组成

7.3.6 TCP/IP的一些扩展特性

另一些Windows 2000的网络服务依赖一些附加驱动程序，扩展了TCP/IP协议驱动程序的基本网络特性，这些利用专用接口的附加驱动程序与TCP/IP协议集成在一起。这些服务包括网络地址翻译（NAT）、网际协议安全性（IPSec）以及服务质量（QoS）。

1. 网络地址翻译

网络地址翻译（NAT）是一个路由服务，它允许多个本地IP地址映射成单个IP地址。如果没有NAT，局域网中的每台计算机必须分配一个公共的IP地址用于因特网通信。NAT使得局域网中的某一计算机可以分配一个公共IP地址，而其他计算机通过那台计算机访问因特网。NAT完成局域网地址与公共IP地址之间的翻译，并且将分组从因特网路由至指定的局域网中的计算机。

Windows 2000上的NAT组件由一个与TCP/IP栈相连接的NAT设备驱动程序和一个管理员用于定义地址翻译的编辑器组成。NAT能够作为一种协议被安装在带路由及远程访问的MMC插件上，或者是在使用网络及拨号连接工具来配置因特网连接共享时，将NAT的协议安装在Windows 2000上（但是当安装了路由及远程访问MMC插件时，NAT能够更好地配置）。

2. 网际协议安全性

网际协议安全性（IPSec）与Windows 2000 TCP/IP栈集成在一起，它提供对IP数据的保护以防止窃听与伪造，并且防御基于IP的攻击。这两个目标通过基于密码技术的保护服务、安全性协议以及动态密钥管理服务得以实现。基于IPSec的通信包括如下一些特点：

- 验证服务确认IP消息的源及其一致性。
- 一致性保证了IP数据在传输过程中不会在无法得知的情况下被篡改。
- 使用加密技术保证了只有消息有效的接收者能够对消息的内容进行解密。

- 反再生性保证了每个分组都是唯一的，而且不可重用。这一特性防止窃听人员针对一个捕获的消息进行回复以达到建立一个会话或者得到未经授权的数据访问许可。

Windows 2000中的IPSec依赖活动目录中配置的组策略，它使用活动目录Kerberos 5的验证方法来验证加入IPSec消息交换的计算机。IPSec使用基于Windows 2000 CryptoAPI，用于加密、解密IPSec消息数据与密码的证书服务的私钥/公钥对。

IPSec的实现由一个IPSec设备驱动程序（\Winnt\System32\Drivers\Ipsec.sys）组成。此驱动程序与TCP/IP协议驱动程序集成在一起。在用户空间内，策略代理从活动目录中获得IPSec的配置信息，然后将IPSec过滤信息（IPSec通信应该使用的IP地址过滤器）传给IPSec驱动程序，并将安全性设置信息传给因特网密钥交换（IKE）模块。IKE模块等待来自IPSec驱动程序安全性关联请求，并仲裁此请求，然后将结果返回给IPSec驱动程序使用。以上过程发生在验证与加密阶段。

3. 服务质量（QoS）

如果没有采用任何特殊的方式，IP通信网提供基于先进先服务策略。应用程序无法控制其消息的优先级，而且会经历突发的网络现象，这种现象指应用程序偶然地获得高吞吐率以及低延迟，然而其他时候它们总是工作在糟糕的网络环境下。尽管这类服务在多数情况下是可以接受的，然而一个网络应用程序数量不断增加的环境需要更可靠的服务级别，或者说需要服务质量（QoS）的保证。视频会议、媒体流和企业资源计划（ERP）都是需要良好的网络性能的几个例子。QoS允许应用程序指定最低带宽和最大延迟，这些特性只要求发送方与接收方的网络软件与硬件支持QoS标准即可，例如IEEE 802.1p，它是定义QoS分组格式与OSI第二层设备（交换机和网络适配器）的响应机制的工业标准。

Windows 2000的QoS是基于微软定义的一些Winsock API，它们使应用程序能够请求在Winsock上通信的QoS。例如，应用程序使用WSCInstallQOSTemplate用于安装QoS模板，它能够指定所需的带宽和延迟（只有拥有管理员权限的应用程序能够使用QoS）。第二个API是通信控制（TC）API，它让管理程序更精确的控制此计算机所连的网络通信量。

Windows 2000 QoS实现的核心是资源预定配置协议（RSVP）Win32服务（\Winnt\System32\Rsvp.exe）。RSVP Winsock服务提供者（\Winnt\System32\Rsvp.dll）通过RPC将应用程序请求传到RSVP服务。随后，RSVP服务使用TC API控制通信量。TC API在\Winnt\System32\Traffic.dll中实现，它将I/O控制命令发送给通用分组分类器（OPC）驱动程序（\Winnt\System32\Drivers\Msgpc.sys）。GPC驱动程序与QoS分组调度器NDIS直通驱动程序通信（\Winnt\System32\Drivers\P Sched.sys），它控制自计算机到网络的分组流，这样所允诺给指定应用程序的QoS级别就能够达到，同时也保证了合适的QoS头能够置于需要QoS的分组内。

7.4 小结

Windows 2000网络架构为网络API、网络协议驱动程序和网络适配驱动程序提供一种灵活的基础设施。Windows 2000网络架构利用I/O层次用于支持网络的扩展性，使得将来的发展能适应计算机网络的发展。当新的协议诞生时，开发人员可以编写TDI传送器在Windows 2000中实现此协议。类似地，新的API能够接入现存的Windows 2000协议驱动程序中。最终，Windows 2000中

实现的网络API的范围能够适应应用程序开发人员的大量可行的实现，每种实现都有其自身的编程模型和协议支持。

习题

- 7.1 请对应OSI模型中的分层结构写出Windows 2000的分层模型。
- 7.2 在Windows 2000中，RPC调用是用户感觉不到的，也就是说，用户进行RPC调用就像LPC调用一样。请问这种对用户的透明性是怎样做到的？RPC与LPC在语义上的差异又表现在哪里？
- 7.3 在图7-12中请就下列几种情况分析操作系统怎样消除数据和控制的不一致性。
 - a) Client1发出Oplock Request后，在收到Level1 grant之前崩溃了。
 - b) Client1收到Level1 grant之后，但在收到Oplock break to none之前崩溃了。
 - c) Client1收到Oplock break to none之后，但在发出Data flush之前崩溃了。

第 8 章

Windows 应用程序设计

第 ⑧ 章

Windows应用程序设计

Windows 应用程序是这样一种应用程序，它是特意为在Windows 环境中运行而编写，并且使用Windows API来完成其任务。本章从操作系统的角度介绍Windows 应用程序设计的有关内容，包括Win32 API、Windows应用程序设计模式、Windows消息机制、结构化异常处理、动态链接库等内容。

需要说明的是，本章重点在于Windows应用程序与操作系统核心之间的关系，而不是介绍用户界面编程。关于Windows用户界面编程，读者可以参阅其他相关的书籍。

8.1 Win32 API

在第2章介绍Windows 2000/XP操作系统体系结构时曾经指出，Windows操作系统分为核心态和用户态两部分。Windows 应用程序总是在常规的用户态下运行，而Windows操作系统核心组件则对外界表现出中立的性质。它们不实现用户界面，甚至不提供编程接口，系统服务调用对应用程序而言是不公开的。那么，应用程序如何利用系统资源、调用系统例程呢？

Windows操作系统依靠一组用户态环境子系统，作为应用程序与操作系统核心之间的接口，环境子系统的主要工作是为应用程序提供编程接口和执行环境。Windows 2000/XP有三个环境子系统：Win32子系统、POSIX子系统和OS/2子系统。Win32子系统是Windows 2000/XP操作系统固有的子系统，这个子系统能够提供应用程序运行所需要的窗口管理、图形设备接口、媒体控制、内存管理等各项服务功能，这些功能以函数库的形式组织在一起，这就是Win32应用程序编程接口（API），简称为Win32 API。Win32子系统负责将API调用转换成Windows操作系统的系统服务调用。

对于应用程序开发人员而言，他所看到的Windows操作系统实际上就是Win32 API，操作系统的其他部分对他来说是完全透明的。从程序员的角度来理解，Windows 应用程序与Win32子系统以及Windows操作系统核心之间的关系如图8-1所示。

Win32 API主要由Win32子系统的三个动态

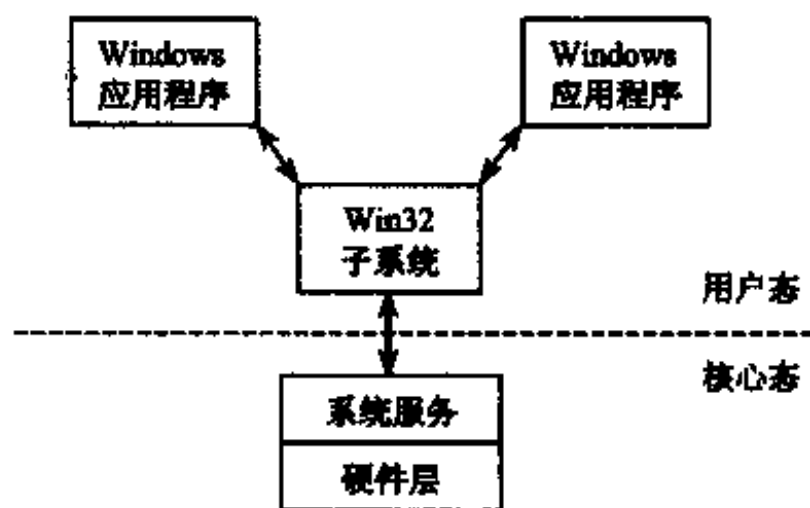


图8-1 Windows 应用程序与操作系统的关系

链接库实现：

- USER32.DLL：负责处理用户接口，包括键盘和鼠标输入、窗口和菜单管理等；
- GDI32.DLL：负责在图形设备（包括显示器和打印机）上执行绘图操作；
- KERNEL32.DLL：操作系统核心功能服务，包括进程与线程控制、内存管理、文件访问等。

除了上述模块以外，Windows 2000/XP还提供了其他一些DLL以支持另外一些功能，包括：通用控件（COMCTL32.DLL）、公共对话框（COMDLG32.DLL）、用户界面外壳（SHELL32.DLL）、图形引擎（DIBENG.DLL）以及网络（NETAPI32.DLL）。

标准Win32 API函数可以分为以下几类：

1) **系统服务**。系统服务函数为应用程序提供了访问计算机资源与底层操作系统特性的手段，包括内存管理、文件系统、设备管理、进程和线程控制等。应用程序使用系统服务函数来管理和监视它所需要的资源。

2) **通用控件库**。系统提供了一些通用控件，这些控件由通用控件库COMCTL32.DLL支持，属于操作系统的一部分，所以它们对所有的应用程序都可用。使用通用控件有助于使应用程序的用户界面与其他应用程序保持一致，同时直接使用通用控件也可以节省开发时间。

3) **图形设备接口**。图形设备接口（GDI）提供了一系列函数和相关的结构，可以绘制直线、曲线、闭合图形、文本以及位图图像等，应用程序可以使用它们在显示器、打印机或其他设备上生成图形化的输出结果。

4) **网络服务**。网络服务函数可以使网络上不同计算机的应用程序之间进行通信。使用网络函数可以创建和管理网络连接，从而实现资源共享，例如共享网络打印机。

5) **用户接口**。用户接口函数为应用程序提供了创建和管理用户界面的方法，可以使用这些函数创建和使用窗口来显示输出、提示用户进行输入以及完成其他一些与用户进行交互所需的工作。大多数应用程序都至少要创建一个窗口。

6) **系统Shell**。Win32 API中包含一些接口和函数，应用程序可使用它们来增强系统Shell各方面的功能。

7) **Windows系统信息**。系统信息函数使应用程序能够确定计算机与桌面的有关信息，例如确定是否安装了鼠标，显示屏幕的工作模式等。

Win32 API是一个基于C语言的接口，但是Win32 API中的函数可以被使用不同语言编写的程序调用，只要在调用时遵循调用规范即可。

不同Windows操作系统平台上的Win32 API存在一些差异，从很大程度上讲，Windows 2000/XP是所有Win32实现的超集。

Win32 API的参考文档可以从<http://www.microsoft.com/msdn>免费获得。

8.2 Windows应用程序设计模式

以窗口为核心的用户界面、以事件驱动为动力的程序运行机制以及将程序代码与用户界面分开处理的程序开发手段，构成了Windows应用程序特有的设计模式。

8.2.1 窗口

作为一个多任务操作系统，Windows的主要设计目标之一是保证用户能够同时访问大多数（如果做不到全部的话）应用程序。如果能给所有应用程序在显示屏上分配一块区域，就能够让用户与所有应用程序打交道。为此在Windows系统中引入了窗口的概念。

窗口是系统显示器上的一个矩形区域，应用程序使用窗口来显示输出或接收用户的输入。一方面应用程序只有通过窗口才能访问系统显示器；另一方面应用程序通过使用窗口与其他应用程序共享系统显示器。同一时间只有一个窗口可以接收用户的输入，用户可以通过鼠标、键盘等输入设备与窗口以及拥有该窗口的应用程序进行交互。

每个Windows应用程序至少要创建一个窗口，称为主窗口，这个窗口是用户与应用程序之间的主要接口。许多应用程序还会直接或间接地创建其他一些窗口，来完成相关的任务。

一旦创建了一个窗口，Windows就能提供该窗口所对应的各种用户交互信息。Windows能够自动完成许多用户请求的任务，如移动窗口、调整窗口大小等。

在Windows环境下允许创建任意数目的窗口，Windows能以各种不同方式来显示信息，并负责为应用程序管理显示屏幕、控制窗口的位置和显示，确保不会有两个应用程序在同一时刻争用系统显示器的同一部分。

应用程序窗口一般由标题栏、菜单栏、工具栏、边框、客户区、滚动条等部件组成。一个基本的应用程序窗口的组成如图8-2所示。

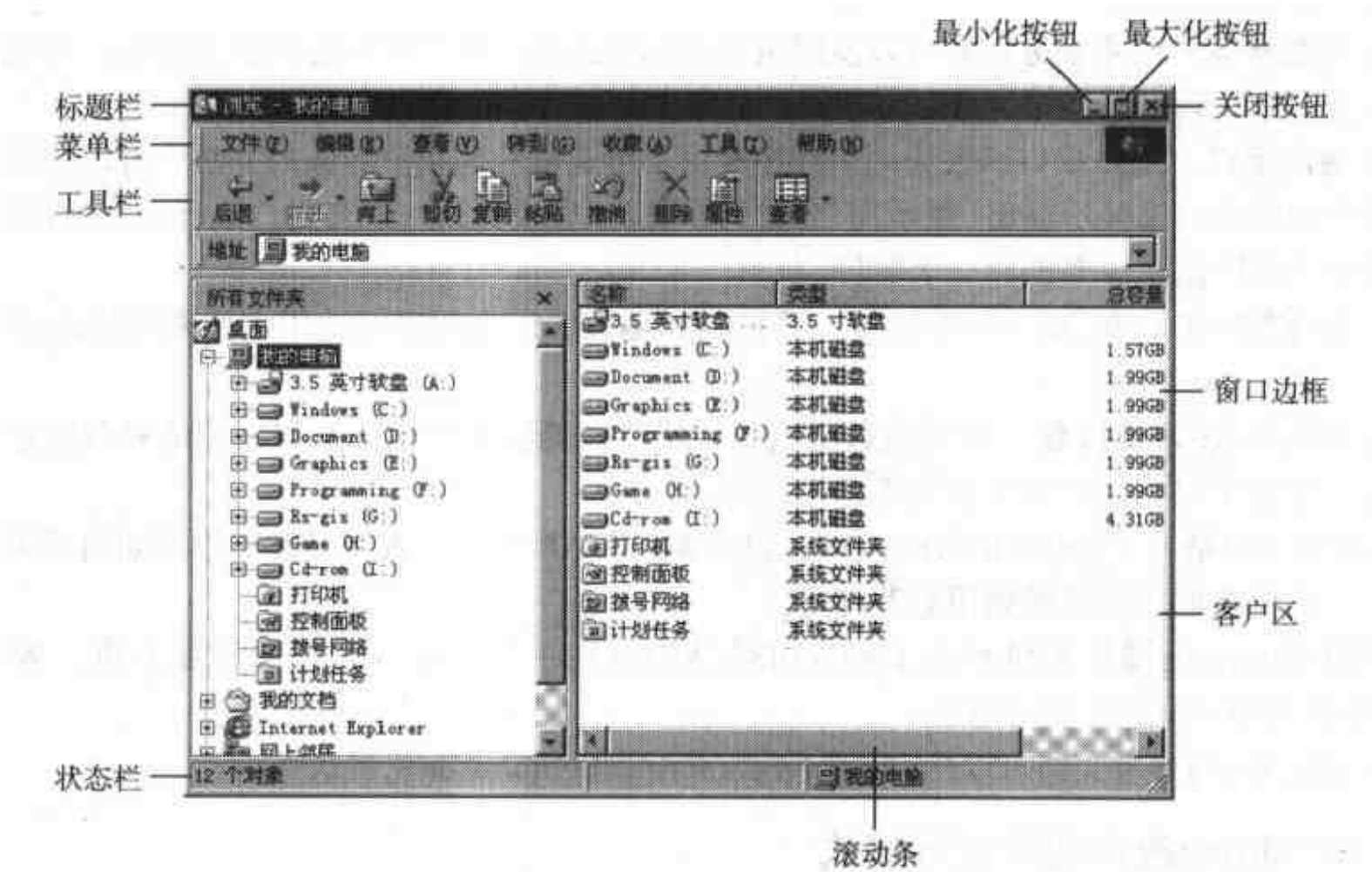


图8-2 Windows应用程序窗口的组成

虽然应用程序创建了某一窗口，并且从技术上来说可以独占它，但该窗口的管理实际上是

由应用程序与Windows操作系统相互协作来实现的。Windows管理窗口的位置和显示方式，并管理窗口的标准部件，如边框、滚动条和标题栏等。而应用程序则管理窗口的其他所有工作，特别是负责管理窗口的客户区（窗口边框以内的区域），应用程序可以完全控制窗口客户区的显示。

应用程序还可以使用另外几种类型的窗口，包括对话框和消息框。对话框是含有一个或多个控件的临时窗口，应用程序可以通过对话框提示用户输入完成某一操作所需要的信息。消息框是用于给用户以提示或警告的窗口。对话框和消息框通常不像应用程序窗口那样具有完全的窗口部件。

8.2.2 事件驱动

Windows应用程序的运行需要依靠外部发生的事件来驱动，描述事件发生的信息称为消息（message）。例如，当用户按下键盘的某个键时，系统就会产生一条特定的消息，标识键盘被按下事件的发生。所谓事件驱动，是指Windows应用程序的执行顺序取决于事件发生的顺序，事件驱动程序设计是围绕着消息的产生与处理而展开的。Windows应用程序在运行时不断获得任何可能的输入消息，进行判断，然后再做适当的处理。

如果把应用程序获得的各种消息分类，则可以分为由硬件设备产生的输入消息和来自Windows系统的窗口管理消息。

应用程序通过输入消息来接受输入，鼠标移动或键盘被按下都将产生输入消息。Windows系统负责监视所有输入设备并将输入消息放入一个先进先出的队列之中，该队列是系统定义的用于临时存储消息的内存块，称为系统消息队列。在Windows环境下同一时间可以运行多个应用程序，每个应用程序都有自己的消息队列，称为应用程序队列。

当用户移动鼠标或敲击键盘时，产生的消息首先进入系统消息队列。接着，Windows从系统消息队列中每次移走一条消息，确定目的窗口，并将消息送入创建该窗口的应用程序的消息队列之中。应用程序通过应用程序队列来接收输入，它通过一个称为消息循环的控制结构从应用程序队列中检索消息，并将检索到的消息发送给相应的窗口，由该窗口的窗口函数负责对消息进行判断，并进行相应的处理。这一过程如图8-3所示。

窗口管理消息与输入消息不同，Windows直接将它发送给有关窗口函数，而不通过系统消息队列和应用程序队列。Windows系统通过这种方式直接将影响某窗口的事件通知窗口。例如，当用户激活一个应用程序窗口时，Windows系统就会向该窗口直接发送相应的消息。

需要说明的是，除了上述通常的消息传递机制以外，Windows系统与应用程序还可以通过调用PostMessage函数将消息直接指派到一个应用程序的应用程序队列中，应用程序也可以通过调用SendMessage函数将消息直接发送给一个应用程序的有关窗口函数。

在Windows中，消息用MSG结构来表示，它的定义如下：

```
typedef struct tagMSG
{
    HWND    hwnd;
    UINT    message;
```



```

WPARAM    wParam;
LPARAM    lParam;
DWORD     time;
POINT     pt;
} MSG;
    
```

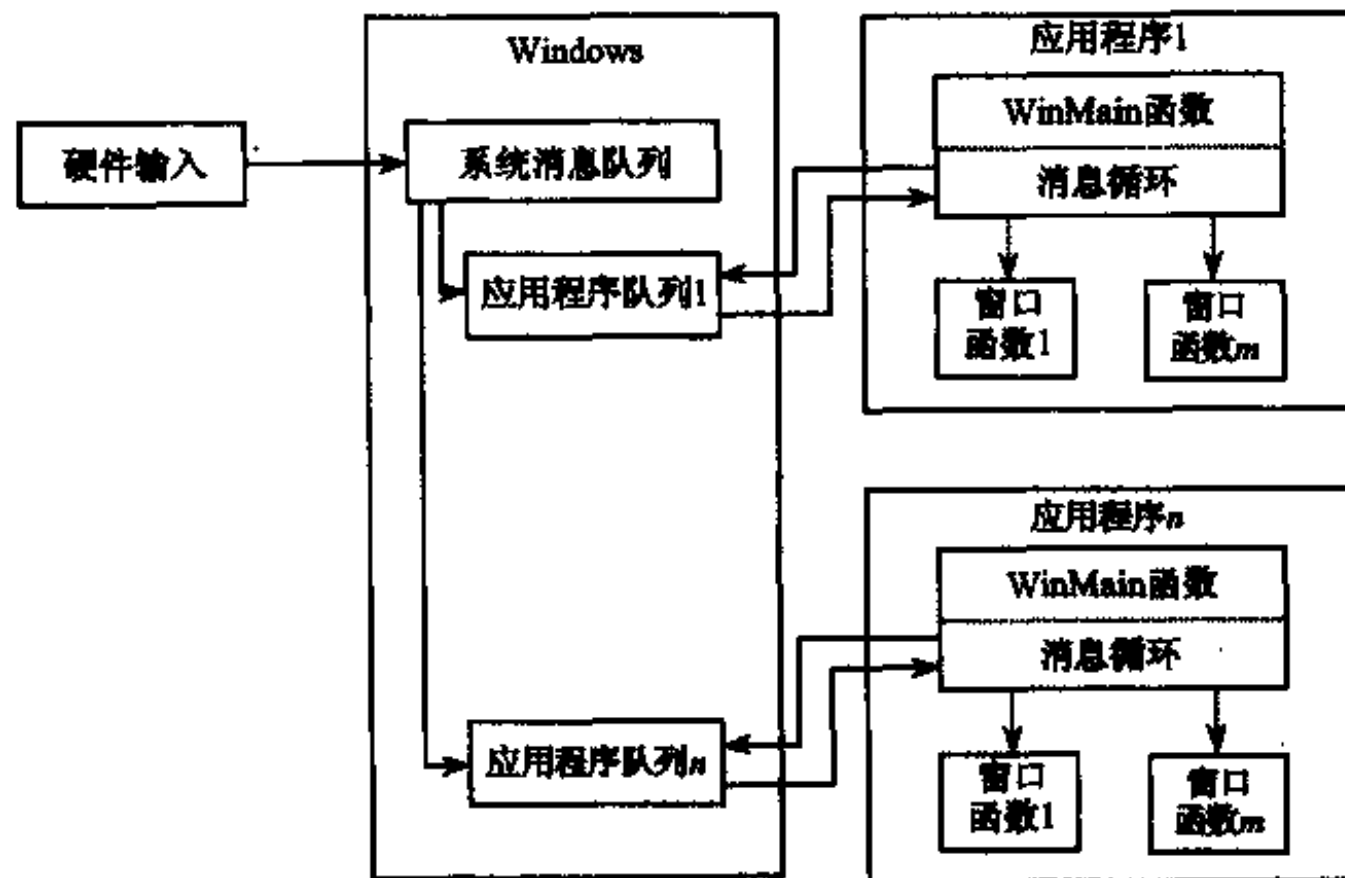


图8-3 Windows中的消息机制

其中：

- hwnd是用于检索消息的窗口句柄，若此参数为NULL，则可检索所有驻留在消息队列中的消息。
- message是代表一个消息的消息标识，每个Windows消息都有一个消息标识。
- wParam和lParam分别为字参数和长字参数，用于提供消息的附加信息。附加信息的含义随不同的消息而有所不同。
- time指定消息送至队列的时间。
- pt指定消息发送时屏幕光标的位置。pt的数据类型POINT也是一个结构，POINT的定义如下：

```

typedef struct tagPOINT
{
    LONG x, y;    //x和y分别表示屏幕的横坐标和纵坐标
} POINT;
    
```

在上述对MSG结构的解释中，涉及句柄、消息标识等与Windows应用程序设计有关的基础性概念。下面对这些概念进行简要说明。

所谓句柄是Windows用来标识被应用程序建立或使用的对象的一个唯一的整数值，Windows

要使用各种各样的句柄来标识诸如应用程序实例、窗口、图标、菜单、输出设备、文件等对象。应用程序通过句柄能够访问相应的对象信息。表8-1是部分常用句柄类型及其说明。

表8-1 常用句柄类型

句柄类型	说 明	句柄类型	说 明
HWND	标识窗口句柄	HDC	标识设备环境句柄
HINSTANCE	标识应用程序实例句柄	HBITMAP	标识位图句柄
HCURSOR	标识光标句柄	HICON	标识图标句柄
HFONT	标识字体句柄	HMENU	标识菜单句柄
HPEN	标识画笔句柄	HFILE	标识文件句柄
HBRUSH	标识笔刷句柄		

在Windows系统中，消息可以分为系统消息和应用程序定义的消息两种类型，每个消息都有一个唯一的消息标识。系统消息标识通常用符号常量来表示，符号常量的定义在Windows的系统头文件windows.h之中。符号常量采用不同的前缀符号来区别能够解释和处理消息的窗口类型。下面是前缀及相应的消息类别：

- BM——表示按钮控件消息
- CB——表示组合框控件消息
- DM——表示默认按钮控件消息
- EM——表示编辑控件消息
- LB——表示列表框控件消息
- SBM——表示滚动条控件消息
- WM——表示窗口消息

8.2.3 Windows应用程序的开发流程

Windows 应用程序分为程序代码和用户界面资源两部分，两部分通过资源编译器组合为一个完整的EXE文件。

用户界面资源包括菜单、对话框、图标、位图、光标、键盘加速键表等，相对于Windows应用程序的程序代码而言，它们属于静态数据。这些资源的实际内容（二进制码）可以借助各种编辑工具制作，并以各种文件扩展名来保存，例如，.ICO为图标文件，.BMP为位图文件，.CUR为光标文件等。程序员必须在一个资源描述文件（.RC文件）中描述它们。资源编译器（RC.EXE）读取资源描述文件的描述，将所有用户界面资源集中制作出一个.RES文件，再与程序代码结合在一起，才能得到一个完整的Windows可执行文件。

将用户界面资源一类的静态数据与程序代码相分离有如下一些优点：

- 减少内存要求。
- 划清了程序员与用户界面设计人员的任务分工。
- 用户界面风格的变化可以不必修改程序代码或只需进行少量的修改。

采用SDK开发Windows 应用程序时，典型的开发流程如图8-4所示。

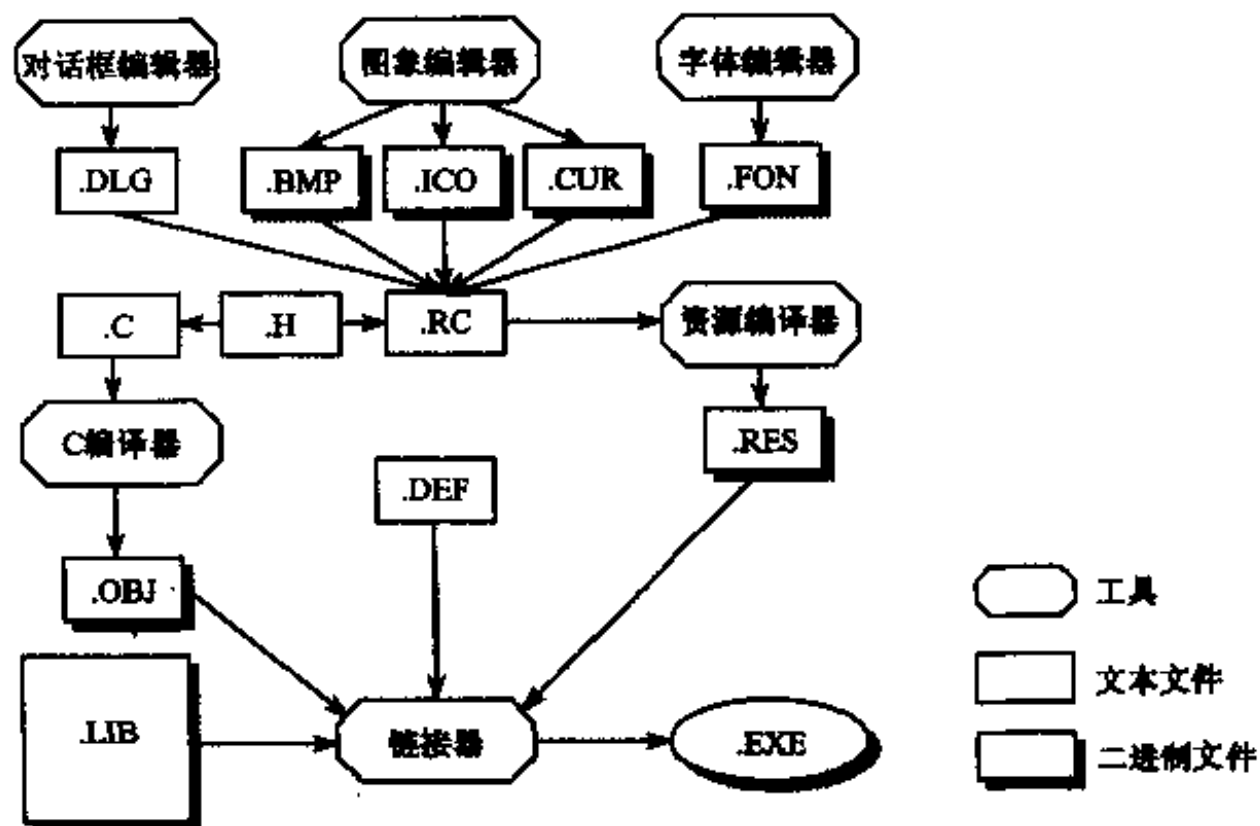


图8-4 Windows 应用程序开发流程

8.3 Windows应用程序的基本结构

Windows应用程序具有相对固定的基本结构，入口点函数WinMain和窗口函数构成了Windows应用程序的基本框架。

8.3.1 WinMain函数

每个Windows 应用程序都必须有一个WinMain函数，它是程序的入口点，相当于标准C语言中的main函数。WinMain函数的说明如下：

```

int WINAPI WinMain
(
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR     lpszCmdLine,
    int       nCmdShow
)

```

这个说明是所有Windows应用程序的标准。当用户开始一个应用程序时，Windows向WinMain函数传递下面参数：

- hinstance 应用程序当前实例句柄，当应用程序启动时由Windows创建
- hPrevInstance 如果应用程序的其他实例正在运行，那么该参数将包含当前实例启动前上一次启动的应用程序实例的hInstance值；否则Windows将该参数值置

- `lpCmdLine` 为NULL
- `nCmdShow` 指向包含命令行参数的字符串的长指针
- `nCmdShow` 应用程序开始执行时窗口的显示方式，它指出当应用程序窗口首次显示时是以正常窗口显示、最大化形式显示还是最小化以图标形式显示。应用程序在调用ShowWindow函数时将nCmdShow的取值传递给该函数

值得注意的是，由于Windows是一个多任务操作系统，所以Windows应用程序有可能并行地多次执行。应用程序的每一次执行即为该应用程序的一个实例，Windows系统使用一个实例句柄来唯一地标识它。应用程序建立的许多资源对于所有应用程序实例都可以使用，因此只要应用程序的第一个实例建立这些资源，以后的所有实例都可以使用而不需要再建立。利用hPrevInstance参数可以检测是否有其他实例存在。

在大多数Windows应用程序中，WinMain函数主要由四部分组成：注册窗口类、创建窗口、显示窗口、建立消息循环。下面一一进行介绍。

1. 注册窗口类

在建立任何窗口之前，必须先注册窗口类。窗口类是用于定义窗口属性的模板，这些属性包括窗口的菜单、背景颜色、光标形状等。窗口类还规定了窗口函数，它处理本类窗口的消息。尽管Windows提供了一些预先定义好的窗口类，但大多数应用程序使用自己定义的窗口类，以便能够控制窗口操作的各个方面。

注册窗口类的方法是先有关该窗口类的信息填充WNDCLASS结构，然后用指向WNDCLASS结构的指针作为参数调用RegisterClass函数。

WNDCLASS结构为Windows提供窗口类的名字、属性、资源和窗口函数等信息。该结构的定义如下：

```
typedef struct tagWNDCLASS {
    UINT          style;
    LRESULT       CALLBACK lpfnWndProc();
    int           cbClsExtra;
    int           cbWndExtra;
    HINSTANCE     hInstance;
    HICON         hIcon;
    HCURSOR       hCursor;
    HBRUSH        hBackground;
    LPSTR         lpstrMenuName;
    LPSTR         lpstrClassName;
} WNDCLASS;
```

其中：

- `style` 指定窗口类的风格，为一个按位或二进制标志
- `lpfnWndProc` 指向窗口函数的指针
- `cbClsExtra` 指定在窗口类之后分配的额外字节数，以便为额外的数据保留空间，通常为0

- **cbWndExtra** 指定在窗口实例之后分配的额外字节数，以便为额外的数据保留空间，通常为0
- **hInstance** 指定正在注册的应用程序实例句柄
- **hIcon** 指定窗口的图标
- **hCursor** 指定窗口内使用的光标
- **hBackground** 指定用来着色窗口背景的图刷
- **lpszMenuName** 指向菜单资源名的指针
- **lpszClassName** 指向窗口类名的指针

要注册窗口类，必须设置WNDCLASS结构的各个字段，然后调用RegisterClass函数。例如：

```
WNDCLASS wndclass;  
wndclass.style = CS_HREDRAW | CS_VREDRAW ;  
wndclass.lpfnWndProc = WndProc ;  
wndclass.cbClsExtra = 0 ;  
wndclass.cbWndExtra = 0 ;  
wndclass.hInstance = hInstance ;  
wndclass.hIcon = LoadIcon (hInstance, "MYICON") ;  
wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;  
wndclass.hbrBackground = (HBRUSH)GetStockObject (WHITE_BRUSH) ;  
wndclass.lpszMenuName = "MYMENU" ;  
wndclass.lpszClassName = "MyClass" ;  
if (!RegisterClass (&wndclass))  
    return 0 ;
```

如果RegisterClass(&wndclass)调用成功，将返回一个非0值；否则，应该将0返回给Windows，从而结束应用程序的运行。

在上述代码中，二进制标志style被赋予两个类风格标识符的或，每个CS_标识符代表标志的一位。CS_HREDRAW和CS_VREDRAW标识表明，每当窗口的横向或纵向大小改变时，窗口实例将被重绘。在Windows应用程序设计中，二进制标志十分常见，其每一位代表一定的含义，通过按位或可以对若干条件进行组合。

lpfnWndProc字段的取值为WndProc表明窗口函数为WndProc，当Windows发送消息给该窗口时，WndProc将被调用。为把WndProc函数的地址赋给lpfnWndProc字段，该函数必须在赋值语句之前进行说明——可以在源文件的开头说明，也可以在头文件中说明并在源文件的开头用include语句引入。

在上述代码中还涉及到窗口类定义中常用的如下函数。

(1) LoadIcon函数

LoadIcon函数的作用是在应用程序中加载一个图标。该函数的说明为：

```
HICON LoadIcon (HINSTANCE hInstance, LPCTSTR lpIconName)
```

其中，hInstance为图标资源所在的模块句柄，如为NULL，则使用系统预定义图标。lpIconName为图标资源名或系统预定义图标标识名。

(2) LoadCursor函数

LoadCursor函数的作用是在应用程序中加载一个图标。该函数的说明为：

```
HICON LoadIcon (HINSTANCE hInstance, LPCTSTR lpIconName)
```

其中，hInstance为图标资源所在的模块句柄，如为NULL，则使用系统预定义图标；lpIconName为图标资源名或系统预定义图标标识名。

(3) GetStockObject函数

应用程序经常调用GetStockObject函数获取系统提供的背景刷。该函数的说明为：

```
HBRUSH GetStockObject (int nBrush)
```

参数nBrush为系统提供的背景刷的标识名。

2. 创建窗口

使用CreateWindow函数可以创建窗口，该函数的原型为：

```
HWND CreateWindow
(
    LPCTSTR    lpzClassName,
    LPCTSTR    lpzTitle,
    DWORD      dwStyle,
    int        x,
    int        y,
    int        nWidth,
    int        nHeight,
    HWND       hwndParent,
    HMENU      hMenu,
    HINSTANCE  hInstance,
    LPVOID     lpParam
)
```

其中：

- lpzClassName 指定窗口类名
- lpzTitle 指向窗口标题
- dwStyle 指定窗口的样式。例如WS_OVERLAPPEDWINDOW表示普通的重叠式窗口，WS_VSCROLL表示带垂直滚动条的窗口，WS_HSCROLL表示带水平滚动条的窗口，WS_POPUP表示弹出式窗口等。在实际应用中，可以定义组合式的窗口样式，如WS_VSCROLL | WS_HSCROLL | WS_POPUP
- x, y 指定窗口左上角的坐标
- nWidth, nHeight 指定窗口的宽度和高度
- hwndParent 指定该窗口的父窗口句柄，如果没有父窗口，则该参数为NULL
- hMenu 指定窗口主菜单句柄
- hInstance 正在创建窗口的应用程序当前实例的句柄
- lpParam 指向一个传递给窗口的参数值的指针

CreateWindow函数的返回值为该函数创建的窗口的句柄。

3. 显示窗口

尽管CreateWindow函数可以创建窗口，但它不能自动显示窗口，窗口的显示由ShowWindow和UpdateWindow函数实现。

应用程序调用ShowWindow函数在屏幕上显示窗口，该函数的原型为：

```
BOOL ShowWindow(HWND hWnd, int nCmdShow)
```

其中，hWnd为CreateWindow函数返回的窗口句柄；nCmdShow为窗口显示方式，该参数的取值从WinMain函数被调用时的参数中获取。

显示窗口后，应用程序通常要调用UpdateWindow函数更新窗口的客户区，该函数的原型为：

```
BOOL UpdateWindow(HWND hWnd)
```

其中，hWnd为CreateWindow函数返回的窗口句柄。

4. 建立消息循环

Windows应用程序的运行以消息为核心，但是正如前面提到的，Windows并不直接把输入消息发送给应用程序，而是把输入消息送入应用程序的消息队列之中。此外，Windows和其他应用程序也可以将消息指派到应用程序队列中。应用程序必须读取应用程序队列，检索消息并将它们发送出去，以便适当的窗口函数能够处理它们。负责这一任务的便是消息循环。消息循环的格式通常为：

```
MSG Msg;
...
while(GetMessage(&Msg, NULL, 0,0))
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
```

其中，GetMessage函数从应用程序队列检索消息，并把消息复制到消息结构Msg中。该函数的原型为：

```
BOOL GetMessage
(
    LPMSG    lpMsg,
    HWND     hWnd,
    UINT     wMsgFilterMin,
    UINT     wMsgFilterMax
)
```

通过设置参数wMsgFilterMin和wMsgFilterMax可以实现消息过滤，即仅处理消息标识介于wMsgFilterMin和wMsgFilterMax之间的消息。如果两个参数都为0，则不过滤消息。

TranslateMessage函数负责将消息的虚拟键转换为字符信息，DispatchMessage函数负责将消息队列中的消息发送到相应的窗口函数中进行进一步的处理，这两个函数的原型分别为：

```
BOOL TranslateMessage(CONST MSG *lpMsg)
BOOL DispatchMessage (CONST MSG *lpMsg)
```

参数LpMsg为指向记录消息的结构变量的指针。

从上面消息循环的代码可以看出，只有当GetMessage函数返回非零值时，循环才能终止。GetMessage函数检索到WM_QUIT消息时返回非零值，检索到其他消息均返回NULL。

当用户关闭窗口时，Windows系统将把WM_DESTROY消息发送给该窗口的窗口函数。在这种情况下，窗口函数应该使用PostQuitMessage函数将WM_QUIT消息发送到应用程序队列中，这样可以使OetMessage函数检索到WM_QUIT消息，从而结束消息循环，退出应用程序。

8.3.2 窗口函数

每个窗口必须具有一个窗口函数（window function），或称为窗口过程（window procedure）。窗口函数从Windows接收消息，这些消息或者是WinMain函数中通过消息循环发送的输入消息，或者是直接来自Windows系统的窗口管理消息。典型的输入消息有WM_KEYDOWN、WM_KEYUP和WM_MOUSEMOVE等，典型的窗口管理消息有WM_CREATAE、WM_DESTROY和WM_PAINT等。窗口函数必须检查每一条消息，并根据这些消息完成特定的操作。

窗口函数的一般形式如下：

```
LRESULT CALLBACK WndProc
(
    HWND hWnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam
)
{
    ...
    switch(message)
    {
        case ...
            ...
            break;
        ...
        case WM_DESTROY:
            postQuitMessage(0);
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return(0);
}
```

窗口函数是一个回调函数，它由Windows调用，而应用程序并不会直接调用它的窗口函数。Windows调用窗口函数时，传递的参数有四个：hWnd 参数为窗口句柄；message参数定义消息的类型，它通常用在switch语句中，直接处理相应的情况；wParam和lParam参数包含该消息的附加信息。窗口函数通常使用这些参数来完成所请求的操作。

窗口函数的主体是由一系列case语句组成的消息处理程序段，程序员只需根据窗口可能收到的消息在case语句中编写相应的处理代码即可。

如果窗口函数不处理某些消息，则必须把它们传给DefWindowProc函数。DefWindowProc函数是系统默认的处理过程，把消息传给DefWindowProc函数可以保证所有发送给该窗口的消息均得以处理。

图8-5所示为Windows应用程序的消息处理过程。

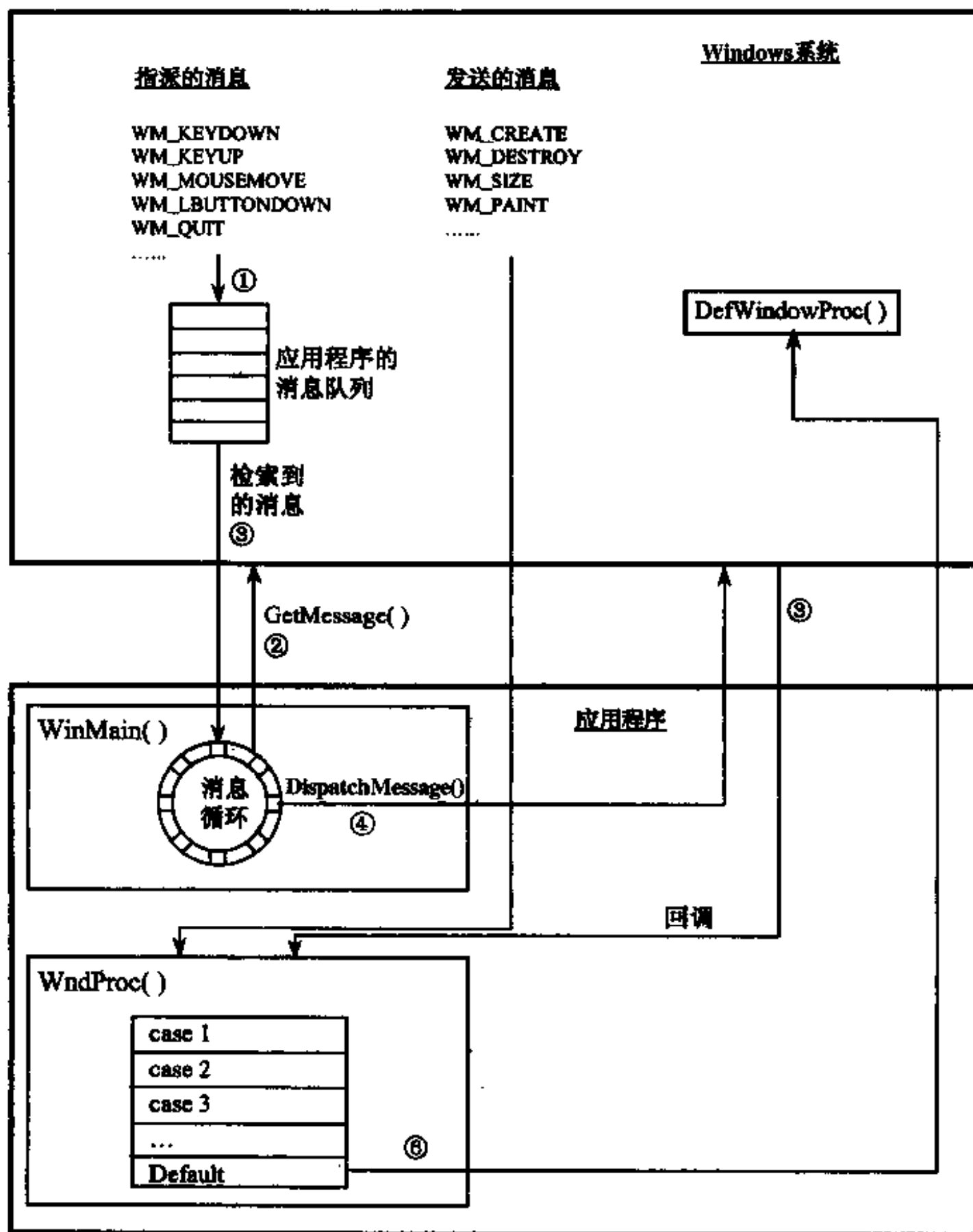


图8-5 Windows应用程序的消息处理过程

在窗口函数的消息处理程序段中，一般都应对WM_DESTROY消息进行处理。如果该窗口函数是应用程序主窗口的窗口函数，则需要通过调用PostQuitMessage(0)把WM_QUIT消息指派到应用程序队列中。如前所述，这样将使WinMain有机会终止消息循环，退出应用程序。图8-6所示为WM_DESTROY消息的处理过程。

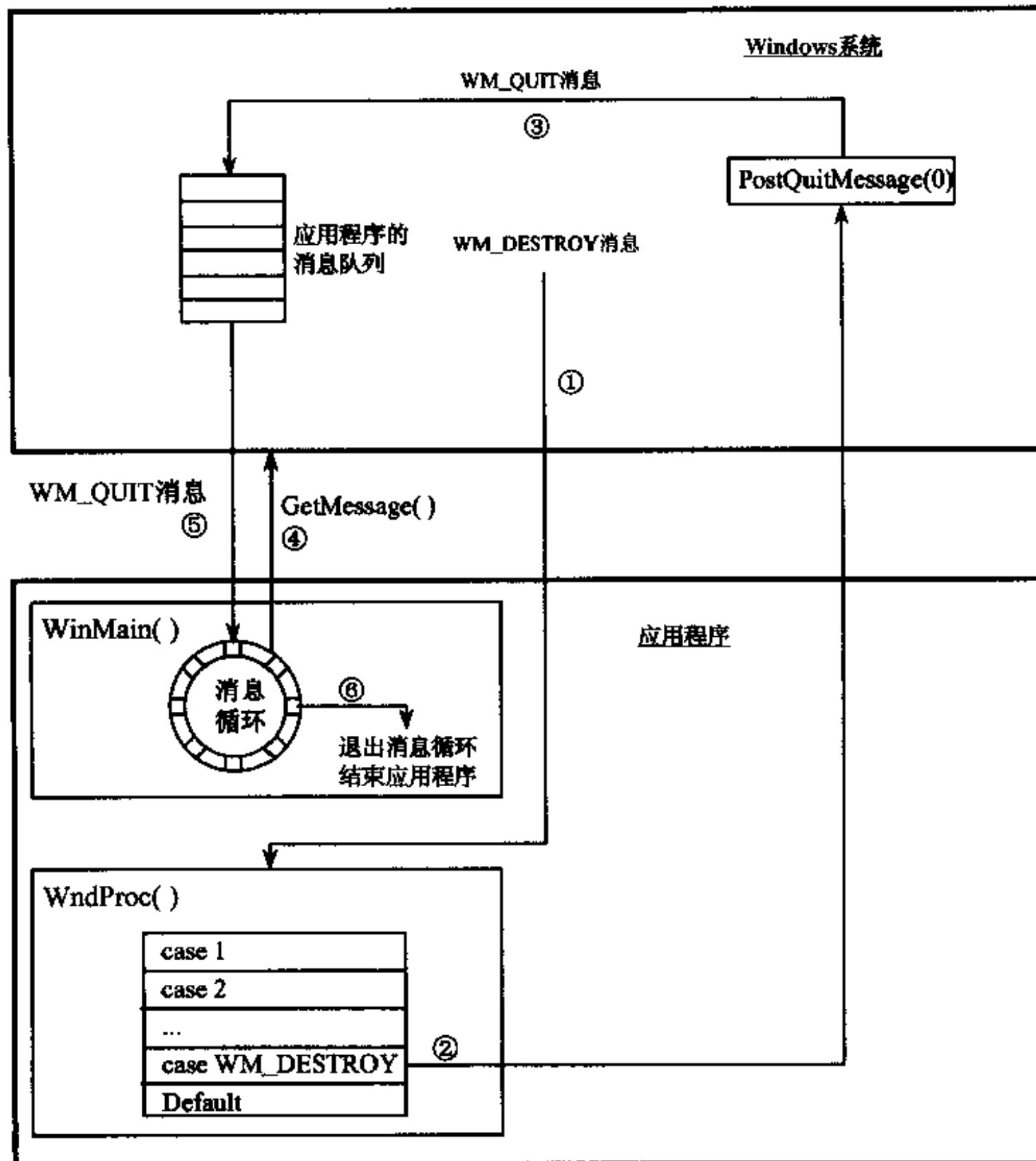


图8-6 WM_DESTROY消息的处理过程

8.4 结构化异常处理

程序在运行时，可能出现各种异常情况，干扰程序的正常执行流程。以往的做法是程序员在

编程时要预计各种异常情况并进行处理，这无疑加重了程序员编程的负担。Windows在系统底层建立了一种称为结构化异常处理（structured exception handling, SEH）的机制，有效减轻了程序员预计并处理各种错误的负担。利用SEH可以把程序主要的工作同错误处理分离开来，这样的分离，可以使程序员集中精力关注程序要完成的任务，而将可能发生的错误放在后面处理。

所谓异常（exception）就是在应用程序的正常执行过程中发生的不正常事件。CPU引发的异常称为硬件异常（hardware exception），例如访问一个无效的内存地址或用0来除一个数值将引发硬件异常。操作系统和应用程序也可以直接引发异常，称为软件异常（software exception）。

SEH是Windows操作系统的一种系统机制，与特定的程序设计语言无关。但是，对于应用程序而言，要利用系统提供的SEH机制，则必须借助于特定程序设计语言的相关语法。因此，SEH不但涉及操作系统，而且与编译器有密切的关系。当异常出现时，编译器要生成特殊的代码，它必须产生一些表来支持处理SEH的数据结构，还要负责准备堆栈结构和其他内部信息，供操作系统使用和参考。编译器还必须提供回调函数，使操作系统可以调用这些函数，保证异常被处理。

如果应用程序不提供异常处理，那么Windows操作系统的异常调度程序将启动系统异常处理程序。系统异常处理程序唯一能做的处理是显示一个描述异常的消息框，然后让用户终止应用程序。编写良好的应用程序应该提供自己的异常处理，这样就有可能从异常中得到恢复，或者至少以一种可控的方式关闭应用程序。

结构化异常处理包括异常处理和终止处理两个方面。

8.4.1 异常处理

需要说明的是，不同的程序设计语言编译器提供的异常处理语法并不相同，这里使用的是Microsoft Visual C++编译器的语法结构。结构化异常处理的语法是：

```
__try
{
    ...           //guarded section
}
__except(exception filter)
{
    ...           //exception handler
}
```

其中，__try和__except是Microsoft Visual C++编译器结构化异常处理使用的关键字。__try所在的代码块称为受保护段（guarded section），它是异常可能发生的地点；__except所在的块称为异常处理程序（exception handler）；__except语句括号中的部分称为异常过滤器（exception filter）。异常过滤器和异常处理程序是通过操作系统直接执行的。

如果受保护段的执行未发生异常，那么except块中的代码永远不会执行；如果受保护段引发了一个异常，系统将定位到except块的开头，并计算异常过滤器的值。异常过滤器可以是一个常数、一个表达式或者一个函数，但是异常过滤器只能返回如下三个异常标识符之一。

(1) EXCEPTION_EXECUTE_HANDLER

这个值告诉异常调度程序执行except块中代码，并且在except块中代码执行完之后，程序在except块后面的代码上恢复执行。这样受保护段中在异常发生处之后的所有代码都将不会再被执行。当except块能够完全修正异常，并且try块中的其他代码没有用处时，应该将异常过滤器的取值设定为EXCEPTION_EXECUTE_HANDLER。例如：

```
DWORD dwTemp = 0;
...
__try
{
    dwTemp = 5 / dwTemp;    //generates an exception
    ...                    //never executes
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
    dwTemp = 0;
}
...                        //start here after exception
```

(2) EXCEPTION_CONTINUE_EXECUTION

这个值告诉异常调度程序，不执行except块而跳回到产生异常的指令试图再次执行。例如：

```
int x = 0;
char *pchBuffer = NULL;
__try
{
    strcpy(pchBuffer, "Hello, world!");
    x = 5 / x;
    ...
}
__except(Filter1(&pchBuffer))
{
}
...

LONG Filter1(char **ppchBuffer)
{
    if(*ppchBuffer == NULL)
    {
        *ppchBuffer = GlobalAlloc(GPTR, 100);
        return(EXCEPTION_CONTINUE_EXECUTION);
    }
    return(EXCEPTION_EXECUTE_HANDLER);
}
```

请读者自行分析上述代码的执行流程。

需要说明的是，程序恢复执行的地方是引起异常的指令，而不是包含异常的高级语言语句，因此使用EXCEPTION_CONTINUE_EXECUTION时必须确保能够真正修正异常情况，必要时需

要检查编译生成的机器代码，否则有可能产生一个无穷循环。例如，对于如下C/C++语句：

```
char *pChar1 = NULL;
*pChar1 = 'H';
```

假设编译器对第二条语句产生的机器指令为：

```
MOV EAX, [pChar1]
MOV [EAX], 'H'
```

那么，第二条指令将产生异常，异常过滤器可以捕获这个异常，修改pChar1的值，并通知系统重新执行第二条指令。但是，由于并未修改EAX的值，这样重新执行第二条指令又会产生异常，从而导致无穷循环。

(3) ECXCEPTION_CONTINUE_SEARCH

这个值告诉异常调度程序去查找前面与一个except块相匹配的try块，并调用这个try块的异常处理器。ECXCEPTION_CONTINUE_SEARCH通常用在当try/except块嵌套使用时，希望在一个较高的级别上将异常处理集中在一起的情形。例如：

```
__try
{
    dwCount1 = 256 / dwTemp1;          //may cause an exception
    __try
    {
        dwCount2 = 65536 / dwTemp2;    //may cause an exception
        ...
    }
    __except(EXCEPTION_CONTINUE_SEARCH)
    {
        ...          //do not handle divide exception
    }
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
    MessageBox(NULL, "Divide by zero", "Fatal Error", MB_OK);
    Return 0;
}
```

如果异常调度程序找不到一个合适的应用程序的异常处理程序时，则异常由Windows的系统异常处理程序来处理。

异常过滤器在确定要返回什么值之前，往往需要知道发生的到底是什么异常，为此可以调用OetexceptionCode函数，该函数返回一个指示发生的异常类型的值。下而是一些典型的异常类型：

- EXCEPTION_ACCESS_VIOLATION 试图访问一个非法的地址
- EXCEPTION_INT_DIVIDE_BY_ZERO 试图用一个整数除以零

- EXCEPTION_FLT_DIVIDE_BY_ZERO 试图用一个浮点数除以零
- EXCEPTION_PRIV_INSTRUCTION 试图执行只在内核模式中允许的指令
- EXCEPTION_STACK_OVERFLOW 堆栈溢出

GetExceptionCode函数只能在异常过滤器中调用，或者在异常处理程序中调用，例如：

```
__try
{
    dwTemp = 256 / dwTemp;    //may cause an exception
    ...
}
__except((GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO)?
    EXCEPTION_EXECUTE_HANDLER:EXCEPTION_CONTINUE_SEARCH)
{
    dwTemp = 0;               //handle the DIVIDE_BY_ZERO exception
}
```

8.4.2 终止处理

标准的Windows应用程序通常包括分配资源，使用这些资源，然后释放它们。由于异常改变了控制的流程，因此很容易导致无法释放在产生异常的代码块中分配的资源。使用终止处理程序可以保证进行这样的清除工作。

终止处理程序的语法结构如下：

```
__try
{
    ...
}
__finally
{
    ...
}
```

对于上述的程序代码，受保护段可能正常结束（执行到结束的花括号处），也可能不正常地终止。不论是哪种情况，操作系统和编译器都可以共同确保finally块都能够被执行，finally块称为终止处理程序（termination handler）。

有两种情况可能使受保护段不正常地结束。一种情况是在try块中执行了下面的控制语句return、goto、break或continue等控制语句。例如：

```
CRITICAL_SECTION CriticalSection;
__try
{
    EnterCriticalSection(&CriticalSection);
    ...
}
```

```
        dwTemp *= 256;
        return(dwTemp);
        ...
    }
    __finally
    {
        LeaveCriticalSection(&CriticalSection);
    }
```

受保护段中的return语句试图退出try块并返回dwTemp变量的内容，但是，在这里退出将导致线程无法释放对临界区的控制权。采用终止处理程序就可以保证当return语句试图退出try块时首先执行finally块中的代码，从而保证线程释放对临界区的控制权。

编译器对上述代码进行编译时，遇到try块中的return语句，需要生成代码将返回值保存在编译器建立的临时变量中，然后生成代码执行finally块中包含的指令。在finally块中的指令执行之后，编译器将临时变量的值取出并从程序中返回。因为程序已在try块中返回，所以任何出现在finally块之下的代码将不再执行。

要完成这些事情，编译器必须生成附加的代码，系统要执行额外的工作，程序的性能将因此而受到影响。因此，应尽量避免在受保护段中执行return、goto、break或continue这些控制语句。

为了帮助避免在try块中使用return语句，Microsoft在其C/C++编译器中增加了一个关键字leave，该关键字会引起跳转到try块的结尾，即跳转到try块的右花括号处。由于控制流自然地由try块中退出并进入finally块，所以不产生系统开销。

另一种使受保护段不正常结束的情况是在try块中发生异常，例如：

```
CRITICAL_SECTION CriticalSection;
__try
{
    EnterCriticalSection(&CriticalSection);
    ...
    dwTemp = 256 / dwTemp;    //may cause an exception
    ...
}
__finally
{
    LeaveCriticalSection(&CriticalSection);
}
```

上述代码在try块中分配了一个临界区资源，同时try块中的除法运算可能会导致异常，但是不论异常是否发生，finally块都能够被执行，从而保证线程释放对临界区的控制权。

无论受保护段是正常结束还是不正常地终止，finally块都将执行。为了确定是哪一种情况引起finally块执行，可以在finally块中调用内部函数AbnormalTermination：

```
BOOL AbnormalTermination()
```

如果控制流正常离开try块自然进入finally块，AbnormalTermination将返回FALSE；否则将返回TRUE。

尽管终止处理程序可以捕获try块非正常终止的大多数情况，但是如果在try块中调用ExitThread或ExitProcess，将立即结束线程或进程，而不会执行finally块中的任何代码。另外，如果其他程序调用TerminateThread或TerminateProcess，进程或线程将死掉，finally块中的代码也不会再执行。因此，虽然没有办法阻止其他程序结束当前的线程或进程，但是可以避免在try块中调用ExitThread或ExitProcess。

一个try块后面可以跟一个except块或一个finally块，但是不能同时跟两个。为了处理异常的同时又有一个终止处理程序，可以在try/except块中嵌套try/finally块，或者反过来也可以。限于篇幅就不在这里展开讨论了。

8.4.3 软件异常

当一个函数执行失败时，传统上要返回一些特殊的值来指出失败，函数的调用者可以检查这些特殊值并采取一种替代的动作。如果这个调用者是被另一个调用者调用的函数，那么它还需要将它自己的失败代码返回给它的调用者。这种错误代码的逐层传递会使源程序变得非常难于编写和维护。果用软件异常则可以解决这些问题。

要引发一个软件异常，只需要调用Raiseexception函数：

```
VOID Raiseexception
(
    DWORD dwexceptionCode;
    DWORD dwexceptionFlags;
    DWORD cArguments;
    CONST DWORD *lpArguments;
)
```

第一个参数dwexceptionCode是标识所引发异常的代码，微软定义了部分异常代码，用户也可以遵循微软的异常代码标准格式定义自己的异常代码。第二个参数dwexceptionFlags必须是0或EXCEPTION_NONCONTINUABLE，前者表示一个可继续（可恢复）的异常，后者表示一个不可继续（致命）的异常。第三个参数cArguments和第四个参数lpArguments用来传递有关所引发异常的附加信息，通常不需要附加的参数，只需对lpArguments参数传递NULL。这种情况下，Raiseexception函数忽略cArguments参数。

下面是一个引发一个软件异常的例子：

```
pMem = GlobalAlloc(GMEM_PTR, 20000);
if(pMem == NULL)
    Raiseexception(FATAL_ERROR_MEM, EXCEPTION_NONCONTINUABLE, 0, NULL);
Strcpy(pMem, "Allocation succeeded");
```

程序捕获软件异常采取的方法与捕获硬件异常完全相同。也就是说，前面介绍的内容可以同样适用于软件异常。

8.5 动态链接库

所谓动态链接库 (DLL), 简单地说就是一个可执行程序模块, 模块中包含了可以被其他应用程序或其他DLL共享的程序代码和资源。前面提到Win32 API函数均是通过DLL形式供Windows应用程序调用的。应用程序也可以创建自己的DLL, 在自己的各个应用程序之间共享代码和资源。

8.5.1 动态链接与静态链接

为了使用一个函数库中的某个函数, 应用程序必须与该库链接起来。库函数可以通过静态链接或动态链接两种方式链接到一个应用程序之中。

静态链接是将应用程序调用的函数直接结合到应用程序映像中的传统方法。在链接时, 所有要用到的函数都从函数库中提取出来, 附加到应用程序可执行映像中。由于每个可执行模块都有它自己的一个函数副本代码在应用程序中被复制, 因此在多任务环境中, 有可能在系统中装入同一个函数的多个副本, 这样就增加了内存要求, 影响了系统的效率。图8-7为静态链接的示意图。

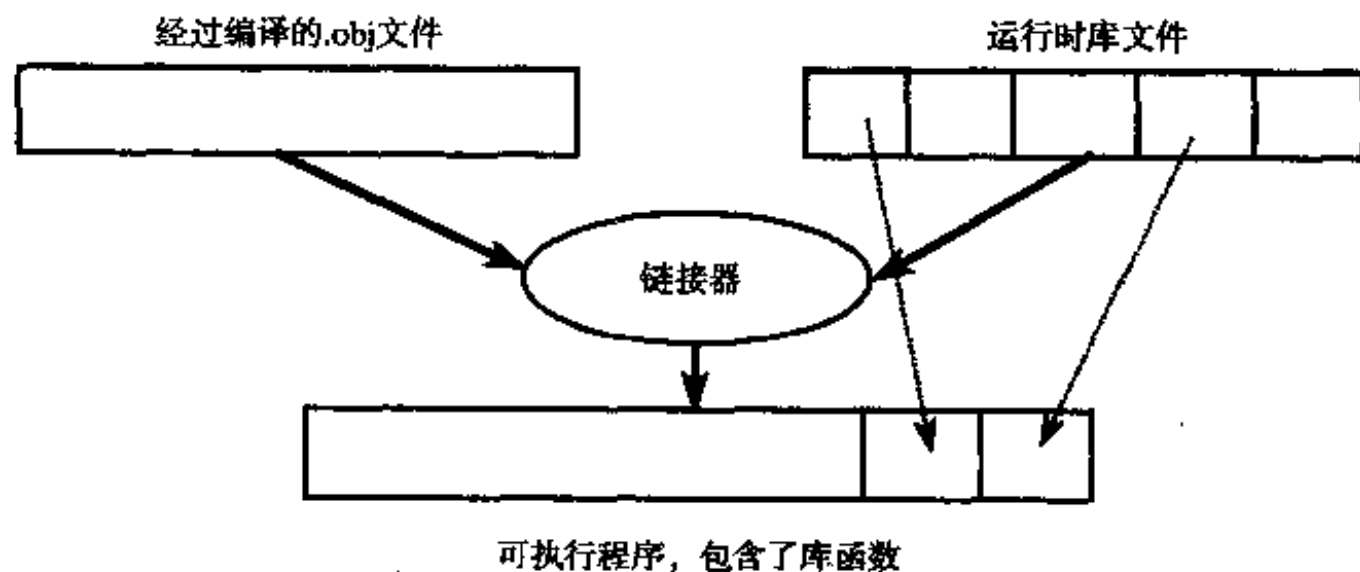


图8-7 静态链接

动态链接指的是在链接时并没有将函数库中的函数链接到应用程序的可执行文件中, 链接是在程序运行时动态地进行的。采用动态链接方式的库文件即为DLL。尽管链接器并不把动态链接的函数复制到可执行文件中, 但是它仍要清楚这些函数在什么地方以及怎样调用它们, 为此需要引入库 (import library) 来帮助链接器使用DLL, 引入库中包含了DLL中函数的重定位信息。图8-8为动态链接的示意图。

采用DLL有许多好处: 当多个进程同时使用同一个DLL时, 只要在内存中装入它的一个副本即可, 从而可以节省内存; DLL与调用它的应用程序相分离, 因此可以在不修改应用程序的情况下对DLL进行更新; 只要在调用DLL中的函数时遵循相同的调用规范, 那么DLL中的函数就可以被各种编程语言编制的程序调用。

当然, DLL也并不能解决所有问题, 它也有一些不足: 使用DLL将增加程序运行时所需的文

件数量，容易引起混乱；当一个依赖于DLL的应用程序在找不到DLL时将无法运行。

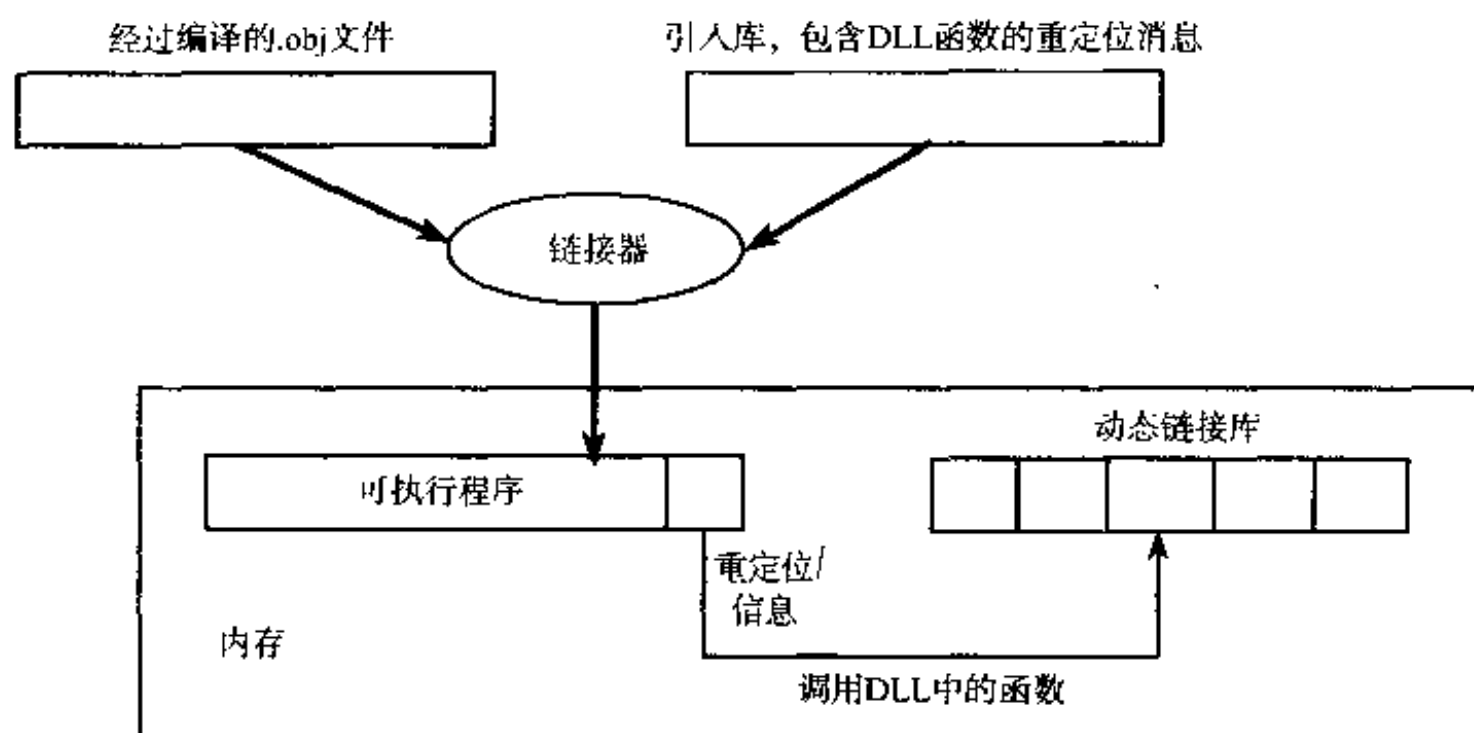


图8-8 动态链接

8.5.2 DLL到进程地址空间的映射

要调用DLL中的函数，首先必须把DLL的文件映象映射到调用进程的地址空间中。有两种方法可以实现这一映射：一种是装入时动态链接（load-time dynamic linking），另一种是运行时动态链接（run-time dynamic linking）。

装入时动态链接是将DLL文件映象映射到调用进程地址空间的最简单的方法。当应用程序被编译时，对一个DLL函数的调用将生成一个外部引用。为解决这一外部引用，在链接时需要将一个DLL引入库（.LIB文件）指定为链接到应用程序上的库，LIB文件中含有DLL文件允许应用程序调用的函数列表。当链接器看到应用程序调用了DLL对应的LIB文件中列出的函数时，就在生成的EXE文件中加入相关信息，指出包含所调用函数的DLL文件名。当应用程序运行时，操作系统在装载应用程序时要查看EXE文件映象的内容，并将所有被引用的DLL文件映象映射到进程的地址空间中。

系统在寻找DLL文件时，按以下目录次序进行搜索：

- 1) 包含可执行应用程序的目录。
- 2) 当前目录。
- 3) Windows的系统目录，使用GetSystemDirectory函数可以返回该目录的路径。
- 4) Windows目录，使用GetWindowsDirectory函数可以返回该目录的路径。
- 5) 列在PATH环境变量中的目录。

如果按上述次序找不到DLL，应用程序即被终止。

一个应用程序可以链接若干个DLL，但在程序启动之前有可能并不知道应该使用哪一个。例如，假设应用程序需要调用打印机驱动程序（为一个DLL）中的一个函数，因为所有打印机驱动

程序都支持标准接口，所以应用程序可以预先知道该函数的名字，但是在程序运行并检查系统配置之前并不知道应该调用哪个打印机驱动程序。如果链接推迟到运行期间，那么正确的DLL就可以判定，然后被动态链接，这便是运行时动态链接的基本思路。

在运行时，通过调用LoadLibrary可以使DLL加载到一个进程的地址空间中。该函数的原型为：

```
HMODULE LoadLibrary(LPCTSTR lpszLibFile)
```

其中，lpszLibFile参数包含DLL文件的名称，同时还可以指定一个路径，如果不指定路径，则系统将按装入时刻动态链接相同的次序搜索目录。如果LoadLibrary成功，则函数返回DLL模块的一个句柄；否则函数返回NULL。

当一个DLL用LoadLibrary显式地加载后，在任意时刻可以调用FreeLibrary函数显式地从进程的地址空间中解除该文件的映射，该函数的原型为：

```
BOOL FreeLibrary(HMODULE hLibModule)
```

hLibModule必须是由LoadLibrary返回的同一模块句柄。

不管有多少进程引用了一个DLL，只有一个副本被装入到内存中。如果在同一个进程中再次调用LoadLibrary，则该进程的DLL引用数加1；而如果调用FreeLibrary，则将DLL引用数减1。如果DLL的引用数降为0，则DLL将从进程的地址空间中解除映射。

为了在运行时从DLL中调用一个函数，需要通过调用GetProcAddress获取函数的地址。该函数的原型为：

```
FARPROC GetProcAddress(HMODULE hModule, LPCTSTR lpszProc);
```

其中，hModule通过调用LoadLibrary获得，lpszProc是所需DLL函数的名称。GetProcAddress的返回值是函数的地址。如果无法定位函数，则返回NULL。

假设DllFunc是A.DLL中的一个函数，则对该函数的调用可以按如下方式进行：

```
//说明DLL函数原型
DWORD DllFunc(DWORD dwParam1, DWORD dwParam2);
HMODULE hLibrary;           //DLL模块句柄
FARPROC lpDllFunc;          //DllFunc函数的地址
DWORD dwReturn;
...
hLibrary = LoadLibrary("A.DLL");
if(hLibrary == NULL)
    return FALSE;
lpDllFunc = GetProcAddress(hLibrary, "DllFunc");
if(hLibrary == NULL)
{
    FreeLibrary(hLibrary);
    return FALSE;
}
dwReturn = (*lpDllFunc)(2000, 2001);
FreeLibrary(hLibrary);
return TRUE;
```

DLL为变量分配的内存存在调用进程的地址空间中。如果有两个进程调用同一个DLL，则所有变量都要被分配两次，即在每个地址空间中都要分配一次。也就是说，尽管两个进程要共享DLL代码的同一个副本，但两个进程均有各自的DLL数据副本。

有时候也需要建立数据的一个公共副本供所有进程共享。例如开发打印机驱动程序时，需要定义一个变量bSusy，表示是否已有打印作业在进行，显然bSusy应该为所有应用程序所共享。图8-9所示为DLL被映射到多个调用进程的地址空间的一般情况。

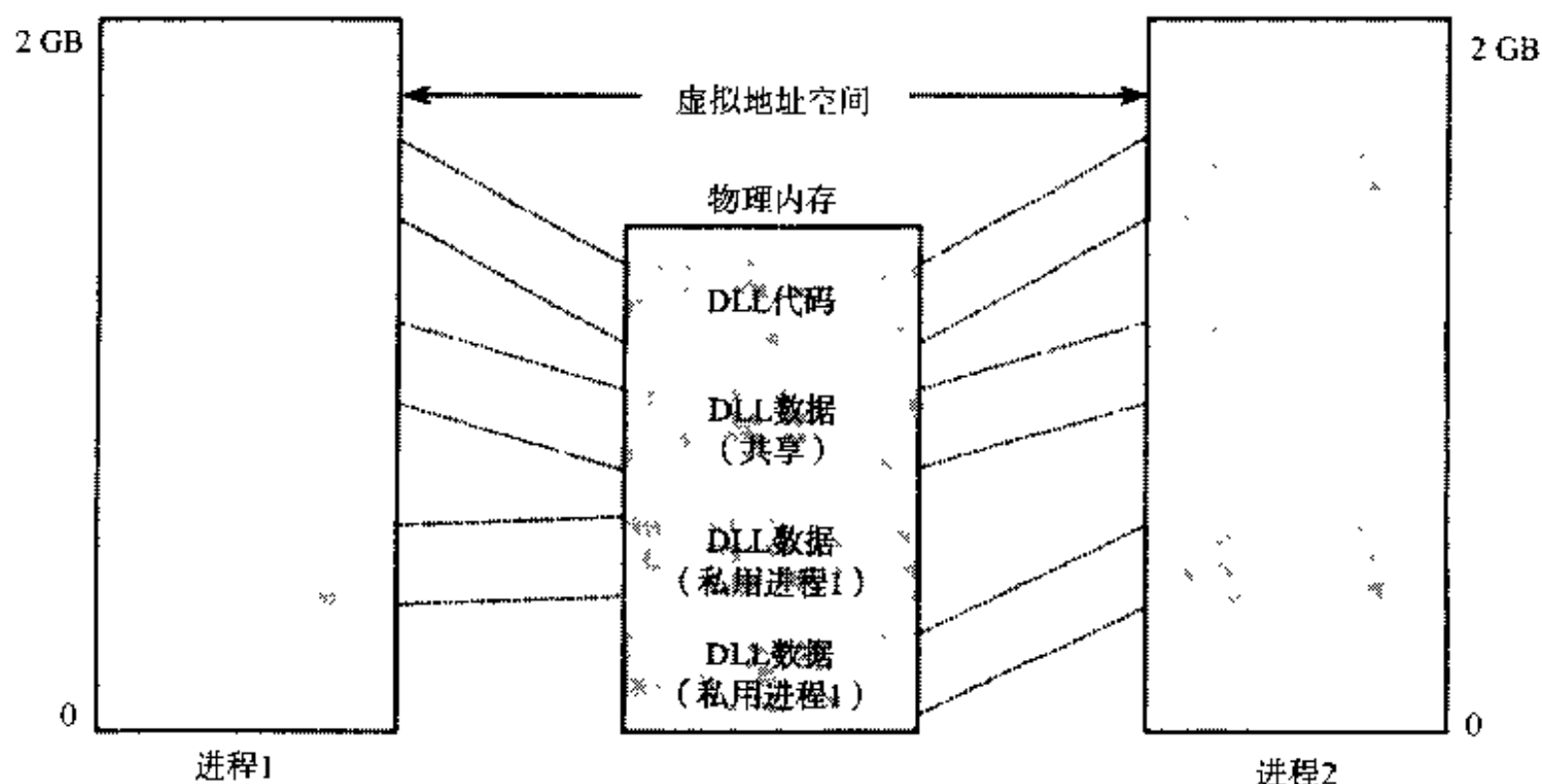


图8-9 DLL与进程的地址空间

8.5.3 DLL的入口点函数

与Windows应用程序不同，DLL没有WinMain函数，不含有消息循环，一般也不获取自己的消息（除非要建立自己的窗口），但是它有自己特殊的入口点函数。DLL入口点函数的缺省名为DllMain。当进程和线程被初始化或终止时，DllMain函数被Windows系统调用。DllMain要做的主要任务是执行进程级或线程级的初始化和清理工作。如果不要求DLL初始化，DllMain可以只是一个虚设函数。

入口点函数DllMain的典型结构如下：

```

BOOL WINAPI DllMain
(
    HMODULE     hModule,
    DWORD       fwdReason,
    LPVOID      lpvReserved
)
{
    ...
    switch(fwdReason)

```

```
{
    case DLL_PROCESS_ATTACH:
        ...
        break;
    case DLL_THREAD_ATTACH:
        ...
        break;
    case DLL_THREAD_DETACH:
        ...
        break;
    case DLL_PROCESS_DETACH:
        ...
        break;
}
return (TRUE);
}
```

入口点函数有三个参数。第一个参数hModule是DLL的模块句柄。有的书上将hModule的类型写为HINSTANCE。对于DLL而言，实例句柄与模块句柄是相同的，但是实例与模块从技术上讲是两个不同的概念。模块由程序文件中的所有对象构成，包括代码、静态变量、数据和资源，这些对象在内存中所采用的形式是模块。实例则包含了模块和当前与它相连的数据的一个副本，程序的所有实例共享一份代码副本，但是与每个实例相连的数据各不相同。

第三个参数lpvReserved当DLL采用装入时动态链接时为非NULL，采用运行时动态链接时为NULL。

第二个参数是dwReason表示调用入口点的原因，其取值为switch语句中的四个值之一。如下所示。

1) DLL_PROCESS_ATTACH。当一个DLL初次映射到进程的虚拟地址空间时，系统调用它的入口点函数，传递的fdwReason参数为DLL_PROCESS_ATTACH。如果线程在后来为已经映射到进程的地址空间中的DLL调用LoadLibrary函数，那么操作系统只是递增DLL的使用计数，并不再次用DLL_PROCESS_ATTACH来调用DLL的入口点函数。

当处理DLL_PROCESS_ATTACH时，DLL应该执行必要的初始化工作，并且DLL应当利用返回值指明DLL的初始化是否已经取得成功。如果成功，则DllMain应该返回TRUE；否则应该返回FALSE。

2) DLL_PROCESS_DETACH。当DLL被从进程的地址空间中解除映射时，系统调用DLL的入口点函数，传递的fdwReason参数为DLL_PROCESS_DETACH。

当处理DLL_PROCESS_DETACH时，DLL应该执行必要的清理工作。

如果进程的终止是因为系统中的某个线程调用了TerminateProcess，那么系统就没有机会用DLL_PROCESS_DETACH来调用DLL的入口点函数，这样映射到进程地址空间中的DLL在进程终止前无法执行必要的清理工作，可能导致数据丢失。

3) DLL_THREAD_ATTACH。当进程创建一个新的线程时，系统调用当前映射到进程地址空

间中的所有DLL的入口点函数，传递的fdwReason参数为DLL_THREAD_ATTACH。这就为DLL在线程一级初始化数据提供了机会。

需要说明的是，如果线程是一个进程的初始线程，那么DLL入口点函数接受的是DLL_PROCESS_ATTACH，而不是DLL_THREAD_ATTACH。

4) DLL_THREAD_DETACH。当一个线程正常终止时，系统调用当前映射到进程地址空间中的所有DLL的入口点函数，传递的fdwReason参数为DLL_THREAD_DETACH。这就为DLL在线程一级清除数据提供了机会。

如果线程的终止是因为系统中的某个线程调用了TerminateThread，那么系统就没有机会用DLL_THREAD_DETACH来调用DLL的入口点函数，这样映射到进程地址空间中的DLL在线程终止前无法执行必要的清理工作，可能导致数据丢失。

8.5.4 DLL的创建和使用

创建DLL文件需要用到源文件(.C)和头文件(.H)。DLL源文件通常包括入口点函数和供应用程序调用的DLL库函数。头文件中含有DLL要导出的所有函数与变量的说明，在编译DLL的源代码时，要包含头文件；在编译调用DLL的应用程序时，也要包含头文件。所谓导出(export)是指DLL中的函数或变量被应用程序调用或访问。

前面提到，如果不要要求DLL初始化，入口点函数DllMain可以只是如下的一个虚设函数：

```
BOOL WINAPI DllMain(HMODULE hModule, DWORD fdwReason, LPVOID lpvReserved)
{
    return TRUE;
}
```

编写DLL库函数和编写普通的C函数没有什么区别，唯一要注意的是在函数和全局变量前要加上__declspec(dllexport)关键字，例如：

```
__declspec(dllexport)
int Sub(int nPara1, int Para2)
{
    return(nPara1 - nPara2);
}
__declspec(dllexport)
int g_nCount = 0;
```

编译器编译上述代码时将在生成的OBJ文件中嵌入额外的信息，供链接器在链接时分析和处理。链接时，当链接器检查到有关导出变量和函数的信息时，将自动产生一个包含DLL导出函数和变量符号的LIB文件，该文件可以供采用装入时动态链接方法链接应用程序时使用。

在应用程序中调用DLL中的函数或访问DLL中的变量时，必须告诉编译器要调用的函数或要访问的变量是在DLL中的，这时可以在函数或变量的说明之前加上关键字__declspec(dllimport)，例如：

```
__declspec(dllimport)
int Sub(int nPara1, int Para2);
```

上述方法适用于Microsoft Visual C++ 6.0开发环境。

习题

- 8.1 Win32子系统与Win32 API的关系是什么？
- 8.2 什么是窗口？它在Windows应用程序中的主要作用是什么？
- 8.3 什么是事件驱动？Windows应用程序为什么采用事件驱动的程序设计方法，而不是像传统DOS应用程序那样采用过程驱动的程序设计方法？
- 8.4 Windows应用程序中消息是如何传递的？
- 8.5 输入消息与窗口管理消息的区别是什么？为什么有这样的区别？
- 8.6 终止处理程序在进程结束时执行，这种说法是否正确？为什么？
- 8.7 判断下面函数的返回值，并说明理由。

```
DWORD Func()  
{  
    DWORD dwTemp = 0;  
    while (dwTemp < 10)  
    {  
        __try  
        {  
            if (dwTemp == 2)  
                continue;  
            if (dwTemp == 3)  
                break;  
        }  
        __finally  
        {  
            dwTemp++;  
        }  
        dwTemp++;  
    }  
    dwTemp += 10;  
    return(dwTemp);  
}
```

- 8.8 与静态链接相比，动态链接有哪些优点？有哪些缺点？
- 8.9 DLL入口点函数的作用是什么？在什么情况下DLL可以不实现入口点函数？

第 ⑨ 章

Windows设备驱动程序设计

第 ⑨ 章

Windows设备驱动程序设计

在开发微机应用系统的过程中，通常会遇到在Windows环境下对具有特定功能的硬件设备（如数据采集卡），进行直接访问与控制的问题。以往在DOS环境下解决这些问题比较简单，但是在Windows环境下，问题就变得复杂了。因为在Windows操作系统下，CPU运行于保护模式，并且统一管理硬件资源，执行于用户态的应用程序代码不能直接访问硬件，而是要通过调用执行于核心态的设备驱动程序提供的各种服务间接地对硬件资源进行访问，这一机制确保了系统的安全。因此，在Windows环境下，开发设备驱动程序是目前涉及计算机硬件设备的开发人员必须面临的问题。

由于设备驱动程序需要与操作系统最底层进行交互，因此不同的操作系统底层结构对应不同的设备驱动程序模型。Windows 2000/XP与Windows 9x内部结构完全不同，因此两种系统的设备驱动程序是不兼容的。由于各种操作系统结构的不同影响了设备驱动程序的兼容性，为此微软公司在1997年提出了一种全新的Windows驱动程序模型（WDM），并在推出Windows 2000操作系统时正式引入了这一技术。WDM以Windows NT 4.0的内部结构为基础，同时引入了Windows 9x的即插即用特性，为存在于Windows 98和Windows 2000操作系统中的设备驱动程序提供了一个统一的参考框架。不仅如此，WDM驱动程序还可以在不修改源代码的情况下经过重新编译后在非Intel平台上运行，因此可以说WDM是一个跨平台的驱动程序模型。

设备驱动程序与操作系统的I/O子系统有着密切关系，本书第6章已经对Windows 2000/XP的I/O子系统以及设备驱动程序的原理进行了详细的介绍。本章将从应用系统开发人员的角度深入探讨Windows 2000/XP操作系统下WDM设备驱动程序开发方法。由于本章内容与第6章密切相关，因此希望读者在学习本章之前，复习一下第6章的内容。

9.1 Windows 2000/XP的设备驱动程序

设备驱动程序是一个软件，它提供一系列控制硬件设备的函数，使用户应用程序能够以一种规范的方式访问硬件，而不必考虑硬件的物理细节。

Windows 2000/XP操作系统中至少有十几种不同的软件组件都称为驱动程序，正如第6章所指出的，可以把这些驱动程序分为用户模式驱动程序和核心模式驱动程序两大类。图9-1是Windows 2000/XP操作系统中存在的各类驱动程序的一种分类体系。

用户模式驱动程序包含了Win32多媒体驱动、支持MS-DOS应用程序的虚拟设备驱动程序



(Virtual Device Driver, VDD) 和其他保护子系统的驱动程序。VDD是一个用户模式部件, 它可以使DOS应用程序访问x86平台上的硬件。VDD通过屏蔽I/O权限掩码来捕获端口访问操作, 它基本上是模拟硬件操作, 这对于那些直接对裸机硬件编程的应用程序特别有用。尽管虚拟设备驱动程序在Windows 98和Windows 2000中名称相同, 但实际上它们是完全不同的概念。为了相互区别, 通常用VDD缩写代表Windows 2000中的虚拟设备驱动程序, 而用VxD缩写代表Windows 98中的虚拟设备驱动程序。

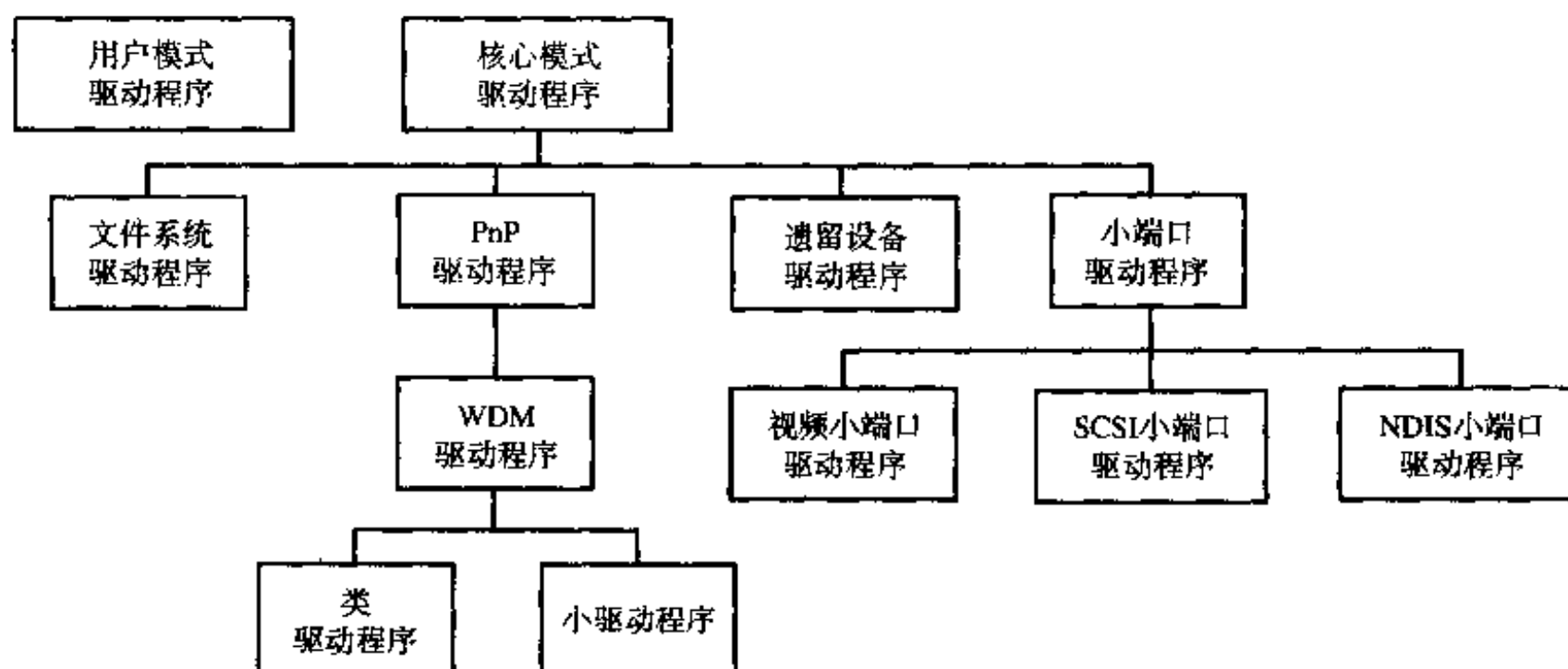


图9-1 Windows 2000/XP设备驱动程序的种类

核心模式驱动程序包含许多子类。

- PnP驱动程序是一种遵循Windows 2000/XP即插即用协议的核心模式驱动程序。
- WDM驱动程序是一种PnP驱动程序, 它同时还遵循电源管理协议, 并能在Windows 98和Windows 2000间实现源代码级兼容。WDM驱动程序还细分为类驱动程序(class driver)和小驱动程序(minidriver), 类驱动程序管理的设备种类已经明确定义, 小驱动程序则向类驱动程序提供厂商特有功能的支持。
- 小端口驱动程序(miniport driver), 包括视频小端口驱动程序、SCSI小端口驱动程序和NDIS小端口驱动程序。
- 文件系统驱动程序(FSD), 在本地硬盘或网络连接上实现标准PC文件系统模型, 包括多层次目录结构和命名文件概念。
- 遗留设备驱动程序(legacy device)是一种不支持PnP的核心模式驱动程序。这种驱动程序主要包括Windows NT早期版本的驱动程序, 它们可以不做修改地运行在Windows 2000中。

从上述分类可以看出, 用户模式驱动程序和遗留设备驱动程序是为了与Windows以前版本兼容而保留的。除此之外, Windows 2000/XP操作系统下的驱动程序开发主要分为三个领域: WDM驱动程序、文件系统驱动程序和小端口驱动程序。其中小端口驱动程序针对的是显示设备、SCSI设备和网络设备等特定应用领域; 文件系统驱动程序针对的是存储设备; WDM驱动程序针

对的则是计算机应用系统开发所面对的大多数情况。限于篇幅，我们不可能展开介绍全部三个领域的驱动程序开发技术，从应用的角度考虑，我们选择WDM驱动程序作为驱动程序开发的一个代表。

9.2 WDM的核心概念和数据结构

WDM是一个分层化的驱动程序模型，在这个模型中，驱动程序的层或堆栈一起工作处理I/O请求。Windows 2000/XP的I/O子系统是基于对象的。对于WDM驱动程序而言，最重要的对象是驱动程序对象和设备对象。Windows 2000/XP的I/O子系统也是一个包驱动的系统。在这个系统中，每个I/O操作可以通过一个IRP描述，驱动程序的工作过程就是对IRP的处理过程。本节从编程的角度深入探讨这些核心概念及相关的数据结构。

9.2.1 设备和驱动程序的分层

WDM模型使用了如图9-2所示的层次结构。图中左边是一个设备对象堆栈，设备对象是系统为帮助软件管理硬件面创建的数据结构。处于堆栈最底层的设备对象称为物理设备对象（Physical Device Object, PDO），在设备对象堆栈的中间某处有一个对象称为功能设备对象（Functional Device Object, FDO），在FDO的上面和下面还会有一些过滤器设备对象（Filter Device Object, FiDO）。位于FDO上面的过滤器设备对象称为上层过滤器，位于FDO下面（但仍在PDO之上）的过滤器设备对象称为下层过滤器。

在WDM驱动程序模型中，每个硬件设备至少有两个驱动程序。其中一个驱动程序称为功能驱动程序，即人们通常所说的硬件设备驱动程序。它了解硬件工作的所有细节，负责初始化I/O操作，处理I/O操作完成时所产生的中断事件，并为用户提供一种适当的设备控制方式。另一个驱动程序称为总线驱动程序（bus driver），它负责管理硬件与计算机的连接。

有些设备除了这两个驱动程序以外还有更多的驱动程序，通常使用过滤器驱动程序（filter driver）这一术语来描述它们。某些过滤器驱动程序仅仅是在功能驱动程序执行I/O操作时进行监视。多数情况下，硬件或软件厂商利用过滤器驱动程序修改现有功能驱动程序的行为。上层过滤器驱动程序在功能驱动程序之前看到IRP，它们有机会提供功能驱动程序根本不知道的额外特征。低层过滤器驱动程序在功能驱动程序要向总线驱动程序发送IRP时看到IRP。在某些情况下，例如当USB设备插入USB总线时，低层过滤器驱动程序可以修改功能驱动程序要执行的总线操作流。

需要说明的是，在WDM驱动程序模型中，总线是一个通用术语，用来描述与设备进行电气连接的硬件。这是一个广义的定义，它不仅包括PCI总线，还包括SCSI卡、并行口、串行口、

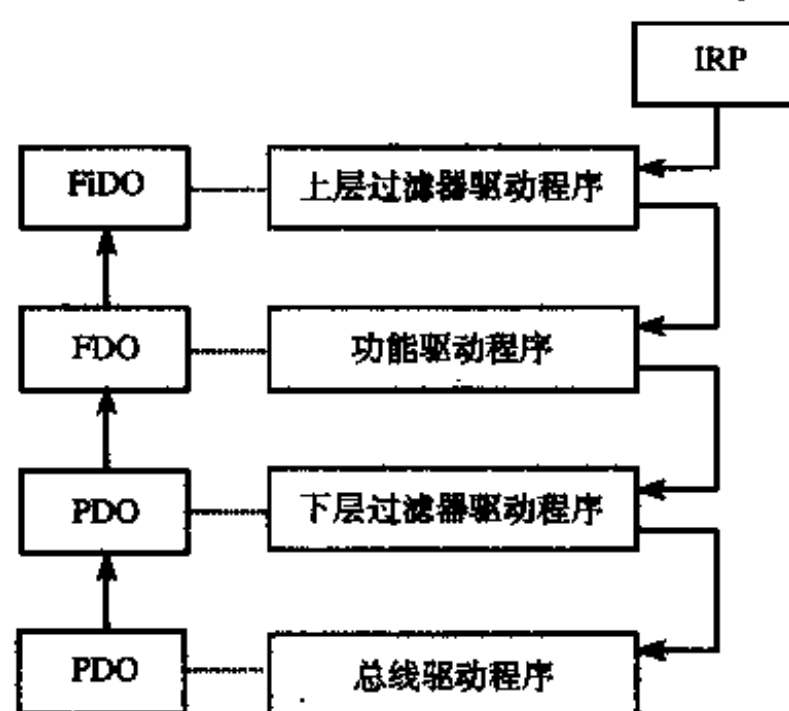


图9-2 WDM中设备对象和驱动程序的分层

USB集线器等，实际上，它可以是任何能插入多个设备的硬件设备。

层次结构可以使I/O请求过程更加明了。每个影响到设备的操作都使用I/O请求包，通常IRP先被送到设备堆栈的最上层驱动程序，然后逐渐过滤到下面的驱动程序。每一层驱动程序都可以决定如何处理IRP。有时，驱动程序不做任何事，仅仅是向下层传递该IRP。有时，驱动程序直接处理完该IRP，不再向下传递。还有时，驱动程序既处理了IRP，又把IRP传递下去。这取决于设备以及IRP所携带的内容。

在单个硬件的驱动程序堆栈中，不同位置的驱动程序扮演了不同的角色。功能驱动程序管理FDO所代表的设备。总线驱动程序管理计算机与PDO所代表设备的连接。过滤器驱动程序用于监视和修改IRP流。由于设备对象与驱动程序软件之间关系紧密，通常可以用FDO驱动程序来代表功能驱动程序，用PDO驱动程序来代表总线驱动程序。

操作系统的PnP管理器按照设备驱动程序的要求构造设备对象堆栈。总线驱动程序的一个任务就是枚举总线上的设备，并为每个设备创建一个PDO。一旦总线驱动程序检查到新硬件存在，PnP管理器就创建一个PDO，之后便开始描绘如图9-2所示的结构。

创建完PDO后，PnP管理器参照注册表中的信息查找与这个PDO相关的过滤器和功能驱动程序，它们出现在图的中部。系统安装程序负责添加这些注册表项，而驱动程序包中控制硬件安装的INF文件负责添加其他表项。这些表项定义了过滤器和功能驱动程序在堆栈中的次序。PnP管理器先装入最底层的过滤器驱动程序并调用其AddDevice函数。该函数创建一个FiDO，这样就在过滤器驱动程序和FiDO之间建立了水平连接。然后，AddDevice把PDO连接到FiDO上，这就是设备对象之间连线的由来。PnP管理器继续向上执行，装入并调用每个下层过滤器、功能驱动程序、每个上层过滤器，直到完成整个堆栈。

在上述过程中，注册表中的三种注册表键起着重要的作用，它们是硬件键（hardware key）、类键（class key）和服务键（service key）。硬件键包含单个设备的信息，类键涉及所有相同类型设备的共同信息，服务键包含驱动程序信息。

设备的硬件键出现在注册表HKEY_LOCAL_MACHINE分支的\SYSTEM\CurrentControlSet\Enum子键上。通常你不能查看到该键的内部信息，系统只允许拥有系统帐号的用户访问该键。即只有内核模式程序和运行在系统帐号下的用户模式服务可以读写Enum键和其子键，但是即使是管理员也不应该直接修改这些键的内容。要想查看Enum键的内容，可以在Administrator特权级帐户下使用REGEDIT32.EXE工具查看。

硬件键中的大部分键值是在安装过程中自动填入的，或者在安装开始后的某个时刻，系统识别出了新硬件（或者经由硬件安装向导）并由系统自动填入的。有些值是由于INF文件中曾指明要放入这里的。我们将在9.4节介绍INF文件。

所有设备类的类键都出现在HKEY_LOCAL_MACHINE 分支的\System\CurrentControlSet\Control\Class键中，它们的键名是由Microsoft赋予的GUID值。

服务键对设备驱动程序而言是一个重要的键，它指出驱动程序执行文件的位置，以及控制驱动程序装入的一些参数。服务键位于HKEY_LOCAL_MACHINE 分支的\System\CurrentControlSet\Services键中。

9.2.2 驱动程序对象

I/O管理器使用驱动程序对象来代表每个设备驱动程序，驱动程序对象描述了驱动程序载入到物理内存的什么地方，驱动程序的大小和它的主要入口点。

在DDK的头文件WDM.H中声明了驱动程序对象的数据结构，其声明为如下形式：

```
typedef struct _DRIVER_OBJECT {  
    CSHORT Type;  
    CSHORT Size;  
    ...  
} DRIVER_OBJECT, *PDRIVER_OBJECT;
```

在声明DRIVER_OBJECT结构的同时还声明了一个PDRIVER_OBJECT指针类型，这种结构声明方式在DDK中大量使用。WDM.H头文件还声明了诸如CSHORT一组类型名，这些类型名用于描述内核模式中的原子数据类型。例如，CSHORT代表用作基数的有符号短整型数。

DRIVER_OBJECT的数据结构如图9-3所示，图中用灰背景表示的一些域是不透明的，这意味着我们仅能直接访问或修改结构中的一部分域。

CSHORT Type	CSHORT Size
PDEVICE_OBJECT	DeviceObject
ULONG	Flags
PVOID	DriverStart
ULONG	DriverSize
PVOID	DriverSection
PDRIVER_EXTENSION	DriverExtension
UNICODE_STRING	DriverName
PUNICODE_STRING	HardwareDatabase
PFastIoDispatch	FastIoDispatch
PDRIVER_INITIALIZE	DriverInit
PDRIVER_STARTIO	DriverStartIo
PDRIVER_UNLOAD	DriverUnload
PDRIVER_DISPATCH	MajorFunction[IRP_MJ_MAXIMUM_FUNCTION+1]

图9-3 DRIVER_OBJECT数据结构

下面我们简要介绍驱动程序对象中主要的透明域。

1) DeviceObject。指向一个设备对象链表，每个设备对象代表一个设备。I/O管理器把多个设备对象连接起来并维护这个域。

2) DriverExtension。指向如图9-4所示的一个子结构，该结构只有AddDevice成员可以直接访问。AddDevice是一个指针，它指向驱动程序中创建设备对象的AddDevice例程。

3) HardwareDatabase。指向一个串，该串为设备的硬件数据库键名。这个注册表键保存着该

设备的资源分配信息。WDM驱动程序没有必要访问该键下的信息，因为PnP管理器自动执行资源分配。

PDRIVER_OBJECT	DriverObject
PDRIVER_ADD_DEVICE	AddDevice
ULONG	Count
UNICODE_STRING	ServiceKeyName

图9-4 DRIVER_EXTENSION数据结构

- 4) FastIoDispatch。指向一个函数指针表，这些函数是由文件系统和网络驱动程序输出的。
- 5) DriverStartIo。指向驱动程序中处理I/O请求的函数。
- 6) DriverUnload。指向驱动程序中的清除函数。WDM驱动程序并没有什么重要的清除工作要做。
- 7) MajorFunction。为一个函数指针表，指向存在于驱动程序中的各个IRP处理函数。它定义了I/O请求如何进入驱动程序。

9.2.3 设备对象

设备对象代表能够成为I/O操作目标的物理设备或逻辑设备，它以DEVICE_OBJECT结构来描述。与驱动程序对象类似，DDK头文件WDM.H中声明DEVICE_OBJECT结构的同时还声明了一个PDEVICE_OBJECT指针类型。图9-5所示为设备对象的格式，其中阴影背景代表不透明域。WDM驱动程序可以调用IoCreateDevice函数创建设备对象，但设备对象的管理则由I/O管理器负责。

CSHORT Type	CSHORT Size
LONG	ReferenceCount
PDRIVER_OBJECT	DriverObject
PDEVICE_OBJECT	NextDevice
PDEVICE_OBJECT	AttachedDevice
PIRP	CurrentIrp
PIO_TIMER	Timer
ULONG	Flags
ULONG	Characteristics
...	...
PVOID	DeviceExtension
DEVICE_TYPE	DeviceType
CCHAR StackSize	...
...	...
SIZE_T	AlignmentRequirement
...	...

图9-5 DEVICE_OBJECT数据结构

下面我们简要介绍设备对象中的透明域。

1) DriverObject。指向与该设备对象相关的驱动程序对象，通常是调用IoCreateDevice函数创建该设备对象的驱动程序对象。过滤器驱动程序有时需要用这个指针来寻找被过滤设备的驱动程序对象，然后查看其MajorFunction表项。

2) NextDevice。指向属于同一个驱动程序的下一个设备对象。这个域把多个设备对象连接起来，起始点就是驱动程序对象中的DeviceObject成员。WDM驱动程序没有必要使用这个域。

3) CurrentIrp。指向最近发往驱动程序StartIo函数的I/O请求包。

4) Flags。包含一组标志位，表9-1列出了其中可访问的标志位。

表9-1 DEVICE_OBJECT结构中的Flags标志

标 志	描 述
DO_BUFFERED_IO	读写操作使用缓冲方式（系统复制缓冲区）访问用户模式数据
DO_EXCLUSIVE	一次只允许一个线程打开设备句柄
DO_DIRECT_IO	读写操作使用直接方式（内存描述符表）访问用户模式数据
DO_DEVICE_INITIALIZING	设备对象正在初始化
DO_POWER_PAGABLE	必须在PASSIVE_LEVEL级上处理IRP_MJ_PNP请求
DO_POWER_INRUSH	设备用电期间需要大电流

5) Characteristics。包含另一组标志位，描述设备的可选特征。I/O管理器基于IoCreateDevice的一个参数初始化这些标志，过滤器驱动程序沿设备堆栈向上方向传播这些标志。表9-2列出了其中可访问的标志位。

表9-2 DEVICE_OBJECT结构中的Characteristics标志

标 志	描 述
FILE_REMOVABLE_MEDIA	可更换媒介设备
FILE_READ_ONLY_DEVICE	只读设备
FILE_FLOPPY_DISKETTE	软盘驱动器设备
FILE_WRITE_ONCE_MEDIA	只写一次设备
FILE_REMOTE_DEVICE	通过网络连接访问的设备
FILE_DEVICE_IS_MOUNTED	物理媒介已在设备中
FILE_DEVICE_SECURE_OPEN	在打开操作中检查设备对象的安全属性

6) DeviceExtension。指向一个由用户定义的数据结构，驱动程序可以使用该结构保存每个设备实例的信息，一般把该结构称为设备扩展（device extension）。I/O管理器为设备扩展对象分配空间，但是设备扩展结构的名称和内容完全由用户决定，常见的做法是把设备扩展结构命名为DEVICE_EXTENSION。设备扩展结构定义的例子见9.3.2节。使用给定的设备对象指针fdo可访问设备扩展结构，代码如下：

```
PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
```

7) DeviceType。是一个枚举常量，描述设备类型。I/O管理器基于IoCreateDevice的一个参数初始化这个成员，过滤器驱动程序有可能需要探测该值。该成员大约有50多种可能值，例如FILE_DEVICE_PRINTER表示打印机，FILE_DEVICE_SCANNER表示扫描仪，FILE_DEVICE_UNKNOWN表示未知设备等。

8) StackSize。统计从该设备对象开始向下直到PDO之间的设备对象个数。该域的目的是告诉其他代码，如果把该设备对象的驱动程序作为其IRP的第一发送对象，那么应在这个IRP中创建多少个堆栈单元。WDM驱动程序通常不需要修改该值，因为创建设备堆栈的支持函数会自动完成这个任务。

(9) AlignmentRequirement。是一个位掩码。执行DMA传输的设备直接使用内存中的数据缓冲区工作。HAL要求DMA传输中使用的缓冲区必须按某个特定界限对齐，而且设备也可能有更严格的对齐需求。AlignmentRequirement域表达了这个约束，它等于要求的地址边界减1。

9.2.4 I/O请求包

正如第6章所指出的，Windows 2000/XP的I/O子系统是一个包驱动的系统，在这个系统中每个I/O操作可以通过一个IRP描述。从编程的角度看，IRP是I/O管理器在响应一个I/O请求时从非分页系统内存中分配的一块大小可变的数据结构内存，I/O管理器每收到一个来自用户的请求就创建一个该结构，并将其作为参数传给驱动程序的DispatchXxx、StartIo等例程。该结构中存放有请求的类型、用户缓冲区的首地址、用户请求数据的长度等信息。驱动程序处理完这个请求后，也在该结构中添入处理结果的有关信息，调用IoCompleteRequest将其返回给I/O管理器，用户程序的请求随即返回。

每一个IRP可以被看成由两部分组成：固定部分和一个I/O堆栈。IRP的固定部分包含关于请求的信息。I/O堆栈则包含一系列I/O堆栈单元（I/O stack location），单元的数目应与驱动程序堆栈中处理这一请求的驱动程序数目相同，每个单元对应一个将处理该IRP的驱动程序。IRP的结构如图9-6所示。

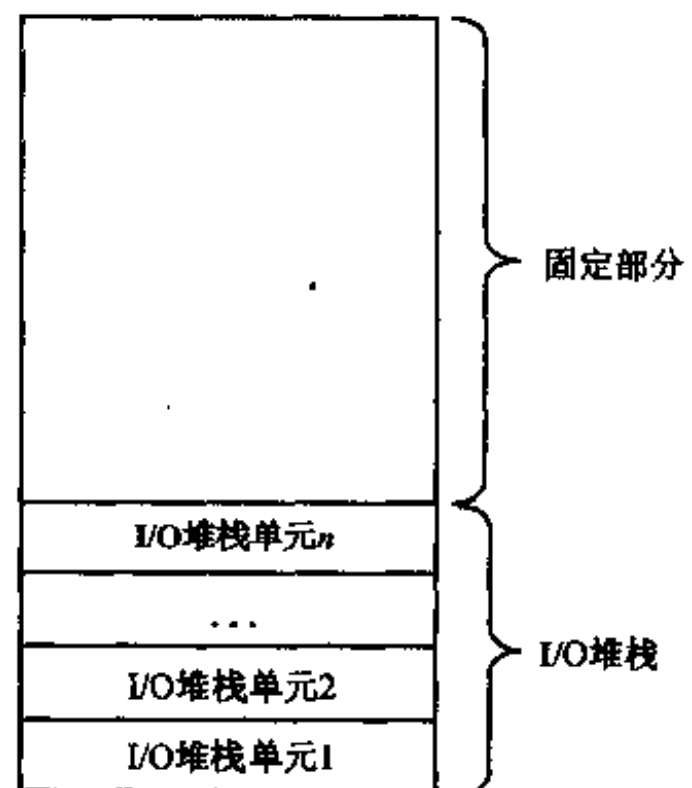


图9-6 IRP的结构

1. IRP固定部分的域

图9-7显示了IRP固定部分的数据结构，下面简要说明该结构中允许驱动程序访问的重要的域。

1) MdlAddress。指向一个内存描述符表（MDL，memory descriptor list）。当驱动程序使用直接I/O时，MDL用来描述一个与该请求相关联的用户模式缓冲区。

2) Flags。包含一些对驱动程序只读的标志。该域通常仅仅是文件系统所关心的，WDM驱动程序与之无关。

3) AssociatedIrp。该域是一个三指针的联合，其中，与WDM驱动程序相关的指针是

AssociatedIrp.SystemBuffer。如果设备执行缓冲I/O，则SystemBuffer指针指向系统空间缓冲区，否则为NULL。

CSHORT Type		CSHORT Size	
PMDL MdlAddress			
ULONG Flags			
union AssociatedIrp			
LIST_ENTRY ThreadListEntry			
IO_STATUS_BLOCK IoStatus			
KPROCESSOR_ MODE RequestorMode	BOOLEAN PendingReturned	CHAR StackCount	CHAR CurrentLocation
BOOLEAN Cancel	KIRQL CancelIrql	CHAR ApcEnvironment	UCHAR AllocationFlags
PIO_STATUS_BLOCK UserIoSb			
PKEVENT UserEvent			
union Overlay			
PDRIVER_CANCEL CancelRoutine			
IO_STATUS_BLOCK UserBuffer			
union Tail			

图9-7 IRP数据结构

4) IoStatus。是一个仅包含两个域的结构，驱动程序在最终完成请求时设置这个结构。IoStatus.Status域将收到一个NTSTATUS代码，而IoStatus.Information的类型为ULONG_PTR，它将收到一个信息值，该信息值的确切含义取决于具体的IRP类型和请求完成的状态。

5) RequestorMode。取值为一个枚举常量UserMode或KernelMode，指定请求初始化的模式为用户模式或核心模式。驱动程序有时需要查看这个值来决定是否要信任某些参数。

6) PendingReturned。该域为一个BOOLEAN类型。如果为TRUE，则表明处理该IRP的最低级分发例程返回了STATUS_PENDING。完成例程通过参考该域来避免自己与分发例程间的潜在竞争。

7) Cancel。该域为BOOLEAN类型。如果为TRUE，则表明IoCancelIrp已被调用，该函数用于取消这个请求。如果为FALSE，则表明没有调用IoCancelIrp函数。

8) CancelIrql。是一个IRQL值，表明那个专用的取消自旋锁是在这个IRQL上获取的。当驱动程序在取消例程中释放自旋锁时应参考这个域。

9) CancelRoutine。指向驱动程序取消例程的地址。应该使用IoSetCancelRoutine函数设置CancelRoutine域而不是直接修改该域。

2. I/O堆栈单元中的域

任何内核模式程序在创建一个IRP时，同时还创建了一个与之关联的I/O堆栈。堆栈中的I/O

堆栈单元由IO_STACK_LOCATION结构定义,每个堆栈单元都对应一个将处理该IRP的驱动程序,如图9-8所示。为了在一个给定的IRP中确定当前IRP的I/O堆栈单元,驱动程序可以调用IoGetCurrentIrpStackLocation函数,该函数返回指向当前I/O堆栈单元的指针。

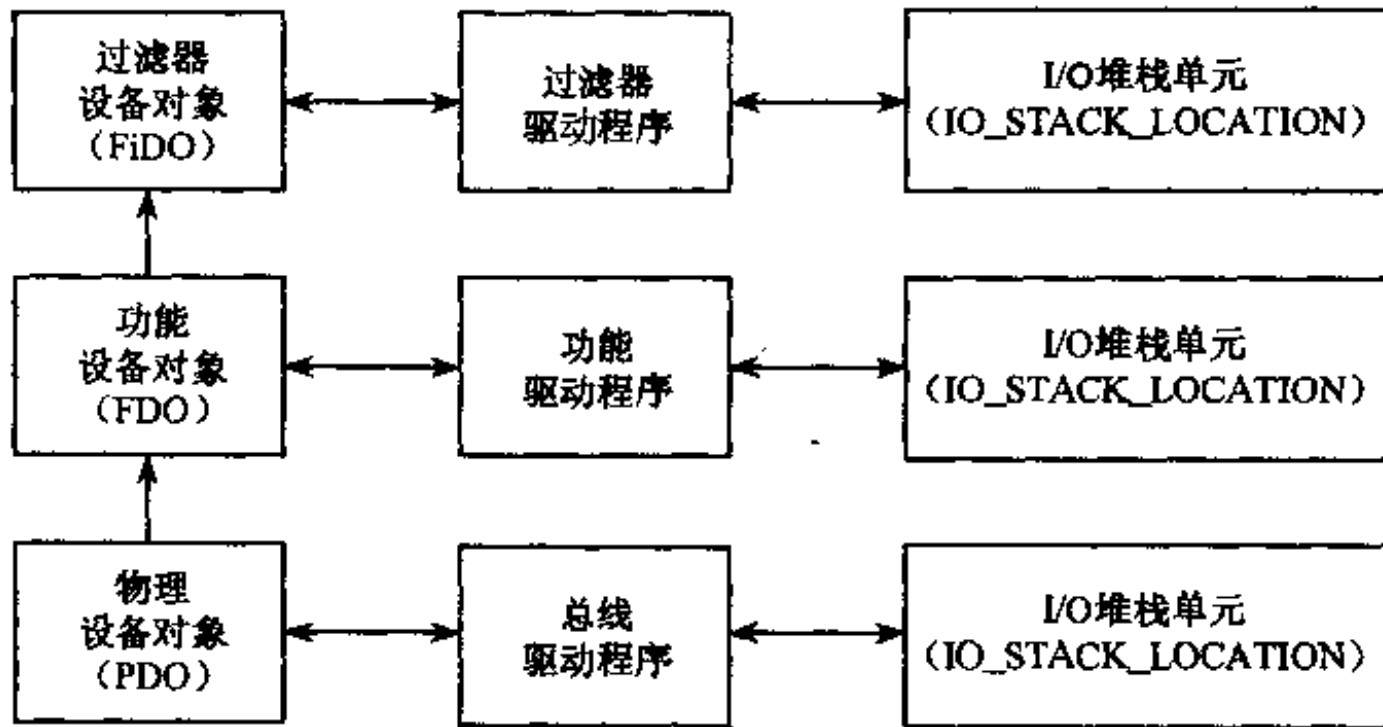


图9-8 驱动程序和I/O堆栈之间的平行关系

当I/O管理器开始分配IRP并且初始化IRP固定部分的时候,它也就初始化IRP中第一个I/O堆栈单元。这个单元中的信息与要传递到驱动程序堆栈中第一个驱动程序的信息相对应,第一个驱动程序将要处理该请求。I/O堆栈单元主要存放I/O请求的函数指针和参数,图9-9显示了I/O堆栈单元的结构。

UCHAR MajorFunction	UCHAR MinorFunction	UCHAR Flags	UCHAR Control
union Parameters			
PDEVICE_OBJECT DeviceObject			
PFILE_OBJECT FileObject			
PIO_COMPLETION_ROUTINE CompletionRoutine			
PVOID Context			

图9-9 I/O堆栈单元数据结构

1) MajorFunction。是该IRP的主功能代码,它指出所要执行的I/O操作类型。例如,主功能代码IRP_MJ_READ表达了通过Win32 API函数CreateFile发布的请求。主功能代码与驱动程序对象的MajorFunction表中的某个分发函数指针相对应。如果该代码存在于某个特殊驱动程序的I/O堆栈单元中,它有可能一开始是IRP_MJ_READ,而后被驱动程序转换成其他代码,并沿着驱动

程序堆栈发送到低层驱动程序。

2) MinorFunction。是该IRP的副功能代码，它进一步指出该IRP属于哪个主功能类。例如，IRP_MJ_PNP请求就有十几个副功能代码，如IRP_MN_START_DEVICE、IRP_MN_REMOVE_DEVICE等。

3) Parameters。它是几个子结构的联合，每个请求类型都有自己专用的参数。这些子结构包括Create（对应IRP_MJ_CREATE请求）、Read（对应IRP_MJ_READ请求）、StartDevice（对应IRP_MJ_PNP的IRP_MN_START_DEVICE子类型）等。

4) DeviceObject。指向该堆栈单元对应的设备对象的地址，该域由IoCallDriver函数负责填写。

5) FileObject。指向与一个I/O请求有关的文件对象的地址。

6) CompletionRoutine。是一个I/O完成例程的地址。该地址是由与这个堆栈单元对应的驱动程序的更上一层驱动程序设置的，驱动程序不要直接设置这个域。

7) Context。是一个任意的与上下文相关的值，将作为参数传递给完成例程。

3. I/O功能代码

Windows 2000/XP使用I/O功能代码辨别将要发生在特定文件对象上的特定I/O操作。I/O功能代码被分为主功能代码和副功能代码，它们都出现在IRP当前I/O堆栈单元中。

每个I/O请求有一个主功能代码并可能有几个副功能代码，主功能代码以IRP_MJ开头的符号定义，副功能代码以IRP_MN开头的符号定义。例如，IRP_MJ_PNP为即插即用IRP主功能代码，IRP_MN_START_DEVICE是表示启动设备的副功能代码。表9-3列出了设备驱动程序经常要用到的主功能代码。

表9-3 常用的IRP主功能代码

功能代码	说 明	对应的Win32 API函数
IRP_MJ_CREATE	打开设备	CreateFile
IRP_MJ_CLEANUP	在关闭设备时，取消挂起的I/O请求	CloseHandle
IRP_MJ_CLOSE	关闭设备	CloseHandle
IRP_MJ_READ	从设备获得数据	ReadFile
IRP_MJ_WRITE	向设备发送数据	WriteFile
IRP_MJ_DEVICE_CONTROL	对用户模式或内核模式客户程序可用的控制操作	DeviceIoControl
IRP_MJ_INTERNAL_DEVICE_CONTROL	只对内核模式客户程序可用的控制操作	没有对应的Win32 API
IRP_MJ_QUERY_INFORMATION	得到文件的长度	GetFileLength
IRP_MJ_SET_INFORMATION	设置文件的长度	SetFileLength
IRP_MJ_FLUSH_BUFFERS	写输出缓冲区或丢弃输入缓冲区	FlushFileBuffers FlushConsoleInputBuffer PurgeComm
IRP_MJ_SHUTDOWN	系统关闭	InitialSystemShutdown

4. IRP的处理

IRP是驱动程序操作的中心。I/O管理器接收一个I/O请求之后，在把它传递到合适的驱动程

序堆栈中的最高层驱动程序之前，分配并初始化一个IRP。

当一个IRP由多个驱动程序处理时，使用多个I/O堆栈单元。每个驱动程序从当前I/O堆栈单元得到它的IRP参数。如果把一个IRP沿当前设备的驱动程序堆栈向上传递，必须使用正确的常数设置下一个堆栈单元。

图9-10说明了一个IRP如何被设备堆栈中的四个驱动程序处理。当IRP到达最高层的驱动程序1时，使用IoGetCurrentIrpStackLocation函数可以获得指向当前堆栈单元的指针。驱动程序1可能做如下工作：处理IRP本身，将IRP传递给较低层次的驱动程序，或者创建一个或更多的附加IRP并将其传递到更低层次的驱动程序。例如，这个IRP可能是最低层驱动程序需要看到的电源管理IRP，驱动程序1对这个IRP可能什么也不做，但它仍需要向上传递这个IRP。

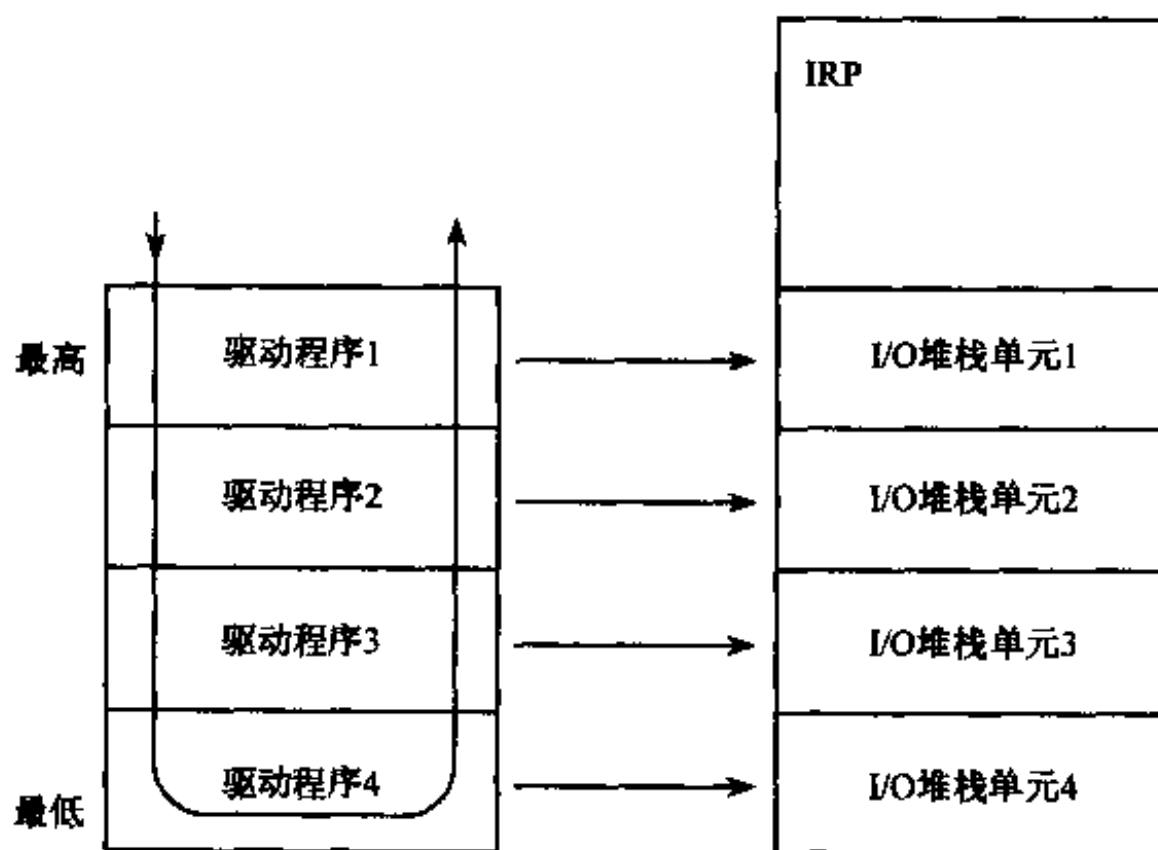


图9-10 IRP的处理

所以驱动程序1需要为下一个驱动程序设置堆栈单元。在多数情况下，它只是使用IoCopyCurrentIrpStackLocationToNext函数或IoSkipCurrentIrpStackLocation函数把当前堆栈单元复制到下一个堆栈单元。如果需要更改下一个堆栈单元，则使用IoGetNextIrpStackLocation函数得到指向这个堆栈单元的指针。

然后驱动程序1使用IoCallDriver函数调用下一个驱动程序，I/O管理器于是改变当前IRP堆栈单元指针，所以驱动程序2看到的是第二个IRP堆栈单元。

继续上述过程，直到最低层的驱动程序4收到这个IRP。

驱动程序4现在处理这个IRP，当它完成或处理时，调用IoCompleteRequest函数，表示它已经完成了这个IRP的处理。IRP再沿设备堆栈向上传递，直到它最终弹出栈顶。

在IRP沿堆栈向上传递时，堆栈中的驱动程序有机会再次处理它。要这样做，驱动程序必须调用IoSetCompletionRoutine函数注册一个完成例程。一个驱动程序不一定要注册完成例程，在

这种情况下，在IRP沿设备堆栈向上传递时，I/O管理器不调用该驱动程序。

驱动程序不一定要沿设备堆栈向下传递IRP，如果它检测到参数中的错误，或者能够处理这个IRP时，则它应该完成它的工作，并使用IoCompleteRequest函数完成该IRP的处理。

9.3 WDM驱动程序的结构

可以把一个完整的WDM驱动程序看作是一个容器，其中包含许多子例程，操作系统调用这个容器中的例程来执行针对IRP的各种操作，图9-11表示了这一概念。



图9-11 WDM驱动程序可执行包中的内容

在每一个WDM驱动程序中，都必须拥有DriverEntry、AddDevice、DispatchPnp、DispatchPower和DispatchWmi这五个例程，其他的例程则是可选的。需要对IRP排队的驱动程序一般都有一个StartIo例程；执行DMA传输的驱动程序应有一个AdapterControl例程；大部分能生成硬件中断的设备，其驱动程序都有一个中断服务例程（Interrupt Service Routine, ISR）和一个推迟过程调用（Deferred Procedure Call, DPC）例程。

此外，WDM驱动程序一般都有几个支持不同类型IRP的分发例程。Windows应用程序与设备驱动程序打交道主要是通过CreateFile、ReadFile、WriteFile和DeviceIoControl等Win32 API来进行的，这些API其实都对应着驱动程序的一些分发例程。实际上，WDM驱动程序除了DriverEntry等必需的例程以外，主要就是由这些分发例程组成的。

WDM驱动程序开发人员的主要任务就是为图9-11所示的容器选择所需要的例程。

9.3.1 DriverEntry例程

驱动程序和一般的DOS/Windows C语言程序不一样，它没有作为入口的main函数或WinMain函数。与DLL相类似，它向操作系统显露一个名称为DriverEntry的函数，在启动驱动程序的时候，操作系统将调用这个入口。

一个驱动程序可以被多个类似的硬件使用，但驱动程序的某些全局初始化操作只能在第一次

被装入时执行一次，DriverEntry例程就是用于这个目的。DriverEntry是每一个设备驱动程序的入口，大部分的设备初始化工作都在这个例程中完成，包括设置响应各种用户请求的分发例程与I/O控制例程的入口，使I/O管理器能知道当用户的打开、关闭、读写等请求到来时各应调用哪些例程来处理。驱动程序中只有本例程的名字DriverEntry是固定的，其他所有例程都要由本例程向系统注册。

如果该驱动程序不响应任何请求的话，只要一个DriverEntry例程就可以构成一个能运行的驱动程序。DriverEntry例程的函数原型为：

```
extern "C" NTSTATUS DriverEntry
(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
)
```

使用extern "C"编译指令的原因是因为有时候我们需要在C代码中使用C++语法，C++语法允许在程序的任何地方声明变量而不仅仅是在花大括号后面。这个预编译指令将禁止编译器生成C++形式的外部函数名修饰，这样连接器就能找到该函数。

DriverEntry函数返回一个NTSTATUS值。NTSTATUS实际上就是一个长整型。大部分内核模式支持例程都返回NTSTATUS状态代码。

函数原型中的IN关键字是在DDK中定义的，表明参数纯粹用于输入目的。除了IN关键字，DDK还定义了OUT和INOUT关键字，其含义一目了然。

DriverEntry的第一个参数是一个指针，指向一个刚被初始化的驱动程序对象。该对象代表的就是当前的驱动程序，DriverEntry例程应完成对这个对象的初始化。

DriverEntry的第二个参数是设备服务键的键名。这个串当函数返回后可能消失，如果以后想使用该串就必须先把它复制到安全的地方。

DriverEntry例程的主要工作是把各种函数指针填入驱动程序对象，这些指针为操作系统指明了驱动程序容器中各种子例程的位置，包括驱动程序对象中的如下指针成员：

1) DriverUnload。指向驱动程序的清除例程。I/O管理器会在卸载驱动程序前调用该例程。通常，WDM驱动程序的DriverEntry例程一般不分配任何资源，所以DriverUnload例程也没有什么清除工作要做。

2) DriverExtension->AddDevice。指向驱动程序的AddDevice函数。PnP管理器将为每个硬件实例调用一次AddDevice例程。由于AddDevice例程对WDM驱动程序特别重要，所以在本章后面还要单独介绍。

3) DriverStartIo。如果驱动程序使用标准的IRP排队方式，应该设置该成员，使其指向驱动程序的StartIo例程。

4) MajorFunction。是一个指针数组。缺省情况下I/O管理器把每个数组元素都初始化成指向一个哑分发函数，该哑分发函数仅返回失败状态码。驱动程序可能需要处理几种类型的IRP，所以至少应该设置与这几种IRP类型相对应的指针元素，使它们指向相应的分发函数。

下面是一个DriverEntry例程的部分代码：

```
extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath)
{
    DriverObject->DriverUnload = DriverUnload;
    DriverObject->DriverExtension->AddDevice = AddDevice;
    DriverObject->DriverStartIo = StartIo;
    DriverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnp;
    DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
    DriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL] = DispatchWmi;
    ...
    servkey.Buffer = (PWSTR) ExAllocatePool(PagedPool,
        RegistryPath->Length + sizeof(WCHAR));

    if (!servkey.Buffer)
        return STATUS_INSUFFICIENT_RESOURCES;
    servkey.MaximumLength = RegistryPath->Length + sizeof(WCHAR);
    RtlCopyUnicodeString(&servkey, RegistryPath);
    return STATUS_SUCCESS;
}
```

在上述代码中，servkey是一个类型为UNICODE_STRING的全局变量，在这里用于备份设备服务键的键名RegistryPath。

9.3.2 AddDevice例程

DriverEntry例程只在驱动程序第一次被装入时执行一次，但是一个驱动程序可以被多个实际设备利用，所以WDM驱动程序有一个特殊的AddDevice函数。PnP管理器为每个设备实例调用该函数。AddDevice函数的原型如下：

```
NTSTATUS AddDevice
(
    PDRIVER_OBJECT DriverObject,
    PDEVICE_OBJECT pdo
)
```

DriverObject参数指向DriverEntry例程中初始化的那个驱动程序对象，pdo参数指向设备堆栈底部的物理设备对象。

对于功能驱动程序，AddDevice函数的基本职责是创建一个设备对象并把它连接到以pdo为栈底的设备堆栈中，主要步骤如下：

- 1) 调用IoCreateDevice创建设备对象，并建立一个私有的设备扩展对象。
- 2) 注册一个或多个设备接口，以便应用程序能知道设备的存在。另外，还可以给出设备名并创建符号连接。
- 3) 调用IoAttachDeviceToDeviceStack函数，把新设备对象放到堆栈上。
- 4) 初始化设备对象的Flag成员。

1. 创建设备对象与设备扩展对象

可以通过调用IoCreateDevice函数创建设备对象，例如：

```
PDEVICE_OBJECT fdo;
NTSTATUS status = IoCreateDevice(DriverObject,
                                sizeof(DEVICE_EXTENSION),
                                NULL,
                                FILE_DEVICE_UNKNOWN,
                                FILE_DEVICE_SECURE_OPEN,
                                FALSE,
                                &fdo);
```

第一个参数DriverObject就是AddDevice的第一个参数，该参数用于在驱动程序和新设备对象之间建立连接，这样I/O管理器就可以向设备发送指定的IRP。

第二个参数是设备扩展结构的大小。I/O管理器自动分配这个内存，并把设备对象中的DeviceExtension指针指向这块内存。

第三个参数是命名该设备对象的UNICODE_STRING的地址，在本例中为NULL。

第四个参数是设备类型，FILE_DEVICE_UNKNOWN表示未知的设备。

第五个参数为设备对象提供Characteristics标志，FILE_DEVICE_SECURE_OPEN表明在打开设备期间执行安全检查。

第六个参数指出设备是否是独占的，对于独占设备，I/O管理器仅允许打开该设备的一个句柄。本例中FALSE表示该设备非独占设备。

最后一个参数&fdo是一个指针，用来保存被IoCreateDevice函数所创建的设备对象的地址。

如果IoCreateDevice由于某种原因失败，则它返回一个错误代码，不改变fdo中的值；如果IoCreateDevice函数返回成功代码，那么它同时也设置了fdo指针。下面的代码对IoCreateDevice返回的NTSTATUS状态代码进行测试：

```
if (!NT_SUCCESS(status))
    return status;
```

如果此后在进行与创建新设备对象相关的其他工作时发现了错误，那么在返回前应先释放刚创建的设备对象并返回状态码。

接下来，我们需要建立一个私有的设备扩展对象。

设备扩展对象的内容和管理全部由用户决定，该结构中的数据成员应直接反映硬件的细节以及对设备的编程方式。例如我们可以这样定义一个设备扩展对象：

```
typedef struct _DEVICE_EXTENSION
{
    PDEVICE_OBJECT DeviceObject;
    PDEVICE_OBJECT LowerDeviceObject;
    PDEVICE_OBJECT Pdo;
    UNICODE_STRING filename;
    WMILIB_CONTEXT WmiLibInfo;
    ...
}
```

```
} DEVICE_EXTENSION, *PDEVICE_EXTENSION
```

其中定义了如下一些域：

1) DeviceObject。我们可以用设备对象中的DeviceExtension指针定位自己的设备扩展对象，同样我们有时也需要在给定设备扩展对象时能定位设备对象，所以这里应该有一个DeviceObject指针。

2) LowerDeviceObject。在调用IoAttachDeviceToDeviceStack函数时，应该把紧接在下面的设备对象的地址保存起来，LowerDeviceObject域用于保存这个地址。

3) Pdo。有一些服务例程需要PDO的地址，而不是堆栈中某个高层设备对象的地址，所以在AddDevice执行时需要在设备扩展对象中保存一个PDO地址。

4) ifname。无论用什么方法来命名设备，都希望能容易地获得这个名字，所以，这里用一个Unicode串成员ifname来保存设备接口名。

设计编写设备驱动程序时，上述结构定义通常在某个头文件中进行。下面的语句创建一个设备扩展对象：

```
PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
pdx->DeviceObject = fdo;
pdx->Pdo = pdo;
```

2. 注册设备接口

通过调用IoRegisterDeviceInterface函数，功能驱动程序的AddDevice函数可以注册一个或多个设备接口，例如：

```
...
status = IoRegisterDeviceInterface(pdo,
                                   &GUID_SIMPLE,
                                   NULL,
                                   &pdx->ifname);
...
```

IoRegisterDeviceInterface的第一个参数必须是设备PDO的地址；第二个参数指出与接口关联的GUID；第三个参数指出额外的接口细分类名称，只有Microsoft的代码才使用名称细分类方案，这里只是简单地将此参数设为NULL；第四个参数是一个UNICODE_STRING串的地址，该串用于接收设备对象的符号连接名。

WDM采用基于GUID的设备命名方案，一个设备接口被一个128位的GUID唯一标识。GUID是globally unique identifier（全局唯一标识符）的缩写。可以用平台SDK中的UUIDGEN工具或者GUIDGEN工具生成GUID。这个想法就像某些工业组织联合起来共同制定某种硬件的标准访问方法一样，在标准制作过程中，产生了一些GUID，这些GUID将永远关联到某些接口上。

如果IoRegisterDeviceInterface由于某种原因失败，则AddDevice函数也是失败。在返回前应先释放IoCreateDevice函数创建设备对象并返回状态码：

```
if (!NT_SUCCESS(status))
{
```



```
IoDeleteDevice(fdo);
return status;
}
```

注册过程实际就是先创建一个符号连接名，然后再把它存入注册表。之后，当响应PnP请求IRP_MN_START_DEVICE时，驱动程序将调用IoSetDeviceInterfaceState函数启用该接口：

```
IoSetDeviceInterfaceState(&pdx->ifname, TRUE);
```

在响应这个调用过程中，I/O管理器将创建一个指向设备PDO的符号连接对象。以后，驱动程序会执行一个功能相反的调用以禁止该接口，即用FALSE做参数调用IoSetDeviceInterfaceState。最后，I/O管理器删除符号连接对象，但它保留了注册表项，即这个名字将总与设备的这个实例关联。

3. 建立设备堆栈

每个过滤器驱动程序和功能驱动程序都有责任把设备对象放到设备堆栈上，可以调用IoAttachDeviceToDeviceStack完成这一工作：

```
...
pdx->LowerDeviceObject = IoAttachDeviceToDeviceStack(fdo, pdo);
...
```

IoAttachDeviceToDeviceStack的第一个参数是新创建的设备对象的地址；第二个参数是PDO地址，AddDevice的第二个参数也是这个地址。

IoAttachDeviceToDeviceStack的返回值是紧接在下面的设备对象的地址，可以是PDO，也可以是其他下层过滤器设备对象。如果该函数失败，则返回一个NULL指针。此时AddDevice函数也是失败的，应返回STATUS_DEVICE_REMOVED：

```
if(!pdx->LowerDeviceObject)
{
    IoDeleteDevice(fdo);
    return STATUS_DEVICE_REMOVED;
}
```

4. 设置设备标志

设备对象中有两个标志位需要在AddDevice中初始化，并且它们在以后也不会改变。它们是DO_BUFFERED_IO和DO_DIRECT_IO标志，但是只能设置并使用其中一个标志，它将决定以何种方式处理来自用户模式的内存缓冲区。例如：

```
fdo->Flags = DO_BUFFERED_IO;
```

在AddDevice中最后一件需要做的事是清除设备对象中的DO_DEVICE_INITIALIZING标志：

```
fdo->Flags &= ~DO_DEVICE_INITIALIZING;
```

当这个标志设置时，I/O管理器将拒绝任何打开该设备句柄的请求或向该设备对象上附着其他设备对象的请求。在驱动程序完成初始化后，必须清除这个标志。

完成上述所有工作后，AddDevice可以返回STATUS_SUCCESS：

```
return STATUS_SUCCESS;
```


9.3.3 DispatchPnp例程

本书第6章中介绍了PnP的理论和Windows 2000/XP的PnP管理器的基本功能。Windows 2000/XP的目标是提供完全的PnP支持，但是具体的PnP支持程度要由硬件设备和相应的驱动程序共同决定。WDM驱动程序必须支持PnP，而支持PnP对驱动程序而言意味着要实现一个AddDevice例程和一个IRP_MJ_PNP处理例程。IRP_MJ_PNP有20多个副功能代码，其中有许多副功能代码仅能由总线驱动程序处理。对功能驱动程序特别重要的是如下三个副功能代码。

1) IRP_MN_START_DEVICE。PnP管理器使用IRP_MN_START_DEVICE来通知功能驱动程序其硬件被赋予了什么I/O资源，以及指导功能驱动程序做任何必要的硬件或软件设置以便设备能正常工作。

2) IRP_MN_STOP_DEVICE。告诉功能驱动程序关闭设备。

3) IRP_MN_REMOVE_DEVICE。告诉功能驱动程序关闭设备并释放与之关联的设备对象。

主IRP_MJ_PNP分发例程一般由基于副功能代码的switch语句组成，一个简化版的IRP_MJ_PNP分发函数看起来像这样：

```
NTSTATUS DispatchPnp(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    ULONG fcn = stack->MinorFunction;
    NTSTATUS status = STATUS_SUCCESS;
    Switch(fcn)
    {
        case IRP_MN_START_DEVICE:
            status = PnpStartDeviceHandler(fdo, irp);
            break;
        case IRP_MN_REMOVE_DEVICE:
            status = PnpRemoveDeviceHandler(fdo, irp);
            break;
        ...
        default:
            PnpDefaultHandler(fdo, irp);
    };
    return status;
}
```

在上述代码中，大部分感兴趣的副功能处理都由调用相应的子例程处理，所有其他的副功能代码在PnpDefaultHandler例程中处理。

9.3.4 DispatchPower例程

从本书第6章对Windows 2000/XP电源管理器的介绍中我们知道，设备驱动程序在电源管理中起着重要的作用。WDM驱动程序必须支持电源管理，这一任务是由DispatchPower分发例程实

现的。当电源管理器发送一个电源IRP（主功能代码IRP_MJ_POWER）时，I/O管理器将调用驱动程序的DispatchPower例程，该例程可以执行以下任务：

- 如果能够处理IRP，则对其进行处理。
- 如果不能处理这一请求，则使用PoCallDriver将IRP传递给驱动程序堆栈中的下一个低层驱动程序。
- 如果是总线驱动程序，则对于设备执行请求的电源管理操作并完成IRP。

关于DispatchPower例程的具体代码，我们将在9.4节的例子中给出。

9.3.5 WMI 与DispatchWmi例程

Windows 2000/XP支持一种称为Windows管理设施（Windows Management Instrumentation, WMI）的控件，用于管理计算机系统。基于Web的企业网管理（Web-based Enterprise Management, WBEM）是一个广泛的工业标准，而WMI是这个工业标准的Microsoft实现。WBEM的核心组件在Windows 2000/XP中是缺省安装的。

图9-12展示了WMI的整体结构。在WMI模型中，数据和事件被分成了消费者和生产者两类。数据块中的内容并不是由WMI指定，而是由数据生产者和数据的使用目的决定。送往驱动程序的数据最有可能来自管理者本身的操作。而驱动程序发出的数据通常是某种性能的统计数据，这些数据的消费者可能是某个性能监视程序。

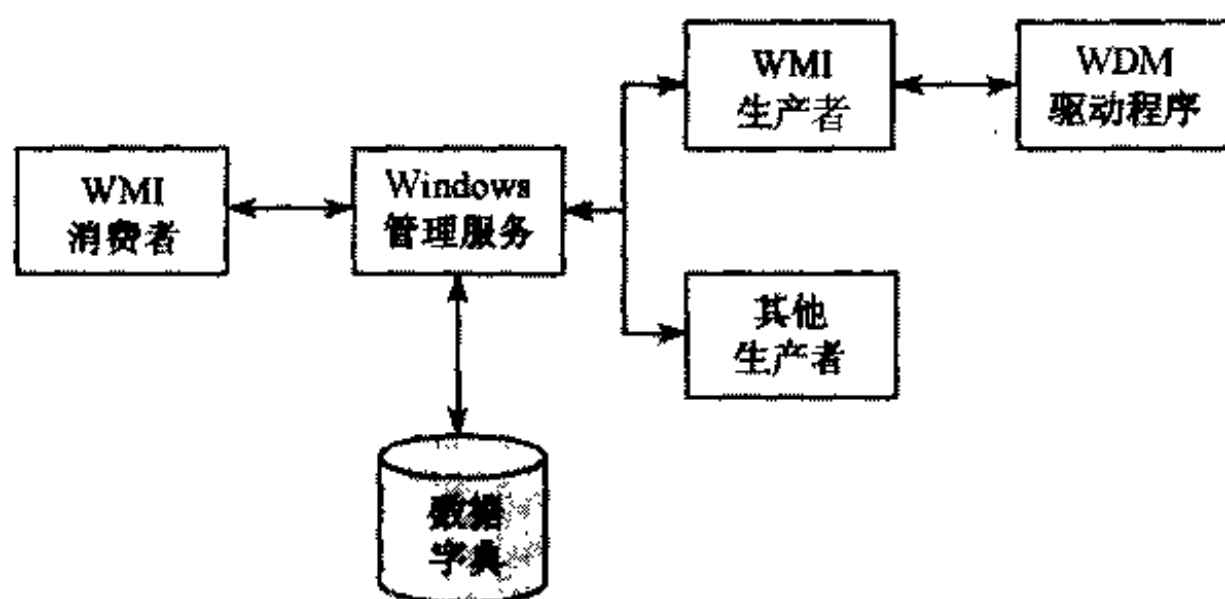


图9-12 WMI的整体结构

在WMI模型中，数据块是抽象类的实例，其概念与C++中的类的概念一致。如同C++中的类一样，WMI类也有数据成员和实现对象行为的方法。WDM驱动程序可以作为WMI类实例的生产者。可以在一个称为模式（schema）的文件中使用一种称为管理对象格式（Managed Object Format, MOF）的语言定义自己定制的数据和事件块。系统则维护一个称为仓库（repository）的数据字典，它包含了所有已知的规划定义。

下面是一个模式文件（.MOF文件）的例子：

```
{Dynamic, Provider('WMIProv'),
```

```
WMI,
Description('Wdm information'),
guid('{87472BA1-61BC-11d2-B677-00C0DFE4C1F3}'),
locale('MS\\0x409')]

class WdmInformation
{
    [key, read]
        string InstanceName;
    [read]
        boolean Active;

    [WmiDataId(1),
        read,
        Description('First ULONG of shared memory buffer')]
        uint32 BufferFirstWord;

    [WmiDataId(2),
        read,
        Description('Symbolic link name')]
        string SymbolicLinkName;
};
```

上述规划表明生产者名为WMIProv，是一个系统部件，它知道该如何实例化这个类。例如，它知道如何调用内核模式例程以及发送一个IRP到适当的驱动程序，它还能根据GUID找到正确的驱动程序。

该规划声明了一个WdmInformation类，该类有四个属性：InstanceName、Active、BufferFirstWord和SymbolicLinkName。在编写驱动程序时，可以用MOF编译器mofcomp编译这个规划并产生一个二进制文件，这个文件最后将成为驱动程序可执行文件的一个资源。在驱动程序初始化过程中，有一部分代码就是告诉WMI生产者这个资源在哪里，以便它能读取这个模式并添入到仓库中。

驱动程序对WMI的支持主要是基于对主代码为IRP_MJ_SYSTEM_CONTROL的IRP的支持。为了能接收到这种IRP，必须先注册这种需求：

```
IoWMIRegistrationControl(fdo, WMI_ACTION_REGISTER);
```

调用IoWMIRegistrationControl函数的恰当位置是在AddDevice例程中。注册完成后，一旦系统认为可以安全地向驱动程序发送系统控制IRP，它就向驱动程序发出一个IRP_MJ_SYSTEM_CONTROL请求，以获得设备的详细寄存信息。

与注册调用作用相反的调用应该在IRP_MN_REMOVE_DEVICE副功能处理函数中：

```
IoWMIRegistrationControl(fdo, WMI_ACTION_DEREGISTER);
```

对于WDM驱动程序而言，系统控制IRP的分发例程DispatchWmi是必须提供的。一般的做法是委托WMILIB来处理系统控制IRP。WMILIB实际上是一个内核模式DLL，它输出的服务可以被其他驱动程序调用。下面是DispatchWmi例程的主要代码：

```
NTSTATUS DispatchWmi(IN PDEVICE_OBJECT fdo, IN FIRP Irp)
{
    PDEVICE_EXTENSION pdx =
        (PDEVICE_EXTENSION)fdo->DeviceExtension;
    SYSCTL_IRP_DISPOSITION disposition;
    NTSTATUS status = WmiSystemControl(&WmiLibInfo,
        fdo, Irp, &disposition);
    switch (disposition)
    {
        case IrpProcessed:
            break;
        case IrpNotCompleted:
            IoCompleteRequest(Irp, IO_NO_INCREMENT);
            break;
        default:
        case IrpNotWmi:
        case IrpForward:
            IoSkipCurrentIrpStackLocation(Irp);
            status = IoCallDriver(pdx->LowerDeviceObject, Irp);
            break;
    }
    return status;
}
```

其中加灰色底纹的语句调用WMILIB来处理系统控制IRP。WmiSystemControl函数的第一个参数是指向一个WMILIB_CONTEXT结构的指针。WMILIB_CONTEXT结构包含8个域。下面的语句定义了一个WMILIB_CONTEXT结构变量WmiLibInfo：

```
WMIGUIDREGINFO GuidList[] =
{
    { &WMI_GUID, 1, 0 },
};
WMILIB_CONTEXT LibInfo;
LibInfo.GuidCount = 1;
LibInfo.GuidList = GuidList;
LibInfo.QueryWmiRegInfo = QueryRegInfo;
LibInfo.QueryWmiDataBlock = QueryDataBlock;
LibInfo.SetWmiDataBlock = SetDataBlock;
LibInfo.SetWmiDataItem = SetDataItem;
LibInfo.ExecuteWmiMethod = NULL;
LibInfo.WmiFunctionControl = NULL;
```

上述代码设置了四个回调函数：

1) QueryRegInfo。在WMI注册之后第一个到来的系统控制IRP带有IRP_MN_REGINFO副功能码。我们把该IRP传递给WmiSystemControl，该函数又回头调用QueryRegInfo函数。

2) QueryDataBlock。当某人想得到你手中的数据时，它们向你发送一个副功能码为IRP_MN_QUERY_ALL_DATA或IRP_MN_QUERY_SINGLE_INSTANCE的系统控制IRP。我们把该IRP委托给WmiSystemControl，该函数然后调用回调函数QueryDataBlock。

3) SetDataBlock。系统会通过发出IRP_MN_CHANGE_SINGLE_INSTANCE请求来要求你改变某个类的整个实例。WmiSystemControl通过调用SetDataBlock回调函数来处理这个IRP。

4) SetDataItem。有时，消费者仅仅想改变WMI对象中的某个域，此时我们将收到一个IRP_MN_CHANGE_SINGLE_ITEM请求，该请求由WmiSystemControl通过调用SetDataItem回调函数来处理。

WmiSystemControl函数返回两个信息：一个NTSTATUS类型的status代码和一个SYSCTL_IRP_DISPOSITION类型的disposition代码。根据disposition代码的不同情况，DispatchWmi例程可以进行相应的处理：如果disposition代码为IrpProcessed，则该IRP已经完成，我们不需要再做任何事情；如果disposition是IrpNotCompleted，则我们有责任完成该IRP。WMILIB已经填充完IRP中的IoStatus块，所以我们仅需要调用IoCompleteRequest；如果不能处理所有可能的特征代码，我们将到达default；如果我们向WMILIB发送一个IRP，但其副功能码在WMI中未定义，则我们到达IrpNotWmi处；IrpForward情况发生于该系统控制IRP是发往其他驱动程序的，因此我们把该IRP下传到下一个驱动程序。

9.3.6 其他例程

前面提到，Windows应用程序与设备驱动程序打交道主要是通过CreateFile、ReadFile、WriteFile和DeviceIoControl等Win32 API来进行的，这些API对应着驱动程序的一些分发例程。例如，当应用程序调用Win32 API CreateFile的时候，操作系统最终转化为对驱动程序IRP_MJ_CREATE功能代码所对应的分发例程的调用。如果驱动程序没有提供该例程，CreateFile调用就会失败。

因此，除了前面介绍的必需的例程以外，WDM驱动程序中通常还包括其他一些例程。下面简要介绍。

1) Unload和ShutDown例程。Unload例程负责在驱动程序被停止前做一些必要的处理，如释放资源，记录最终状态等。ShutDown例程在系统即将关闭时被调用，与前者的区别在于不用释放任何资源。

2) DispatchOpen和DispatchClose例程。这两个例程在用户调用CreateFile和CloseHandle时被调用，为即将到来的读写操作做准备，或做一些读写完成后的必要处理。

3) DispatchRead、DispatchWrite与StartIo例程。前两个例程在用户调用ReadFile和WriteFile时被调用，它们先做一些检验用户请求合法性的工作，然后启动一个称为StartIo的例程，开始与硬件间实际的数据传输。I/O管理器还通过IRP为它们提供了一个指向用户缓冲区的指针，用于与

用户程序交换数据。

正如在9.3.2节介绍DriverEntry例程时所指出的，驱动程序中除了DriverEntry例程必须以DriverEntry命名以外，其他例程都可以使用程序员自定义的名字，并且都要由DriverEntry例程向系统注册。

9.4 WDM驱动程序的编程

前面各节详细介绍了WDM驱动程序的核心概念和基本结构，本节通过一个具体的驱动程序例子WdmDriver介绍WDM驱动程序的基本编程技术。

WdmDriver是一个WDM驱动程序，它实现了一个4字节的共享内存缓冲区，Win32 应用程序可以对该缓冲区进行读写操作。由于采用WDM模型，WdmDriver 可以运行在Windows 98 和 Windows 2000两个平台上。

9.4.1 WdmDriver的源代码组成

WdmDriver的源代码由六个文件组成，包括WdmDriver.h 与guid.h 两个头文件和Init.cpp、Pnp.cpp、Dispatch.cpp与WMI.cpp 四个C++源文件。C++源文件的内容将在后面介绍。下面先看头文件。

(1) WdmDriver.h

WdmDriver.h头文件中包括：函数原型声明；一个UNICODE_STRING类型全局变量servkey的定义，servkey在这里用于备份设备服务键的键名RegistryPath；以及设备扩展结构的定义。下面是WdmDriver.h的部分源代码：

```
#ifndef __cplusplus
extern "C"
{
#endif

#include <wdm.h>
#include <wmilib.h>
#include <wmistr.h>

#include "guid.h"

extern UNICODE_STRING servkey;
typedef struct _WDM_DEVICE_EXTENSION
{
    PDEVICE_OBJECT pdo;
    PDEVICE_OBJECT fdo;
    PDEVICE_OBJECT NextDevice;
    UNICODE_STRING ifSymLinkName;
    WMILIB_CONTEXT WmilibInfo;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

// 以下为函数原型声明
```



```
VOID WdmUnload(IN PDRIVER_OBJECT DriverObject);  
...
```

(2) guid.h

为GUID定义:

```
DEFINE_GUID(WDM_GUID, 0x87472ba0, 0x61bc, 0x11d2, 0xb6, 0x77, 0x0, 0xc0, 0xdf,  
            0xe4, 0xc1, 0xf3);  
DEFINE_GUID(WMI_GUID, 0x87472ba1, 0x61bc, 0x11d2, 0xb6, 0x77, 0x0, 0xc0, 0xdf,  
            0xe4, 0xc1, 0xf3);
```

9.4.2 初始化与清除

源文件Init.cpp实现驱动程序的初始化,其中包括入口例程DriverEntry和清除例程DriverUnload的源代码。

DriverEntry例程的源代码基本上已经在9.3.1节中给出了,其中缺少的语句读者应该能够自行补上。注意在WdmDriver驱动程序的DriverEntry例程中调用ExAllocatePool函数分配了系统空间内存:

```
servkey.Buffer = (PWSTR) ExAllocatePool(PagedPool,  
RegistryPath->Length + sizeof(WCHAR));
```

所以清除例程DriverUnload需要释放该内存,除此之外DriverUnload例程没有什么清除工作要做。下面是DriverUnload例程的源代码:

```
VOID WdmUnload(IN PDRIVER_OBJECT DriverObject)  
{  
    ExFreePool(servkey.Buffer);  
}
```

9.4.3 PnP与电源管理

源文件Pnp.cpp实现了三个例程: AddDevice、DispatchPnp和DispatchPower。

AddDevice例程的大部分源代码在9.3.1节中已经详细解释了,下面把分散的代码组合在一起,写出完整的AddDevice例程:

```
NTSTATUS AddDevice( IN PDRIVER_OBJECT DriverObject,  
                  IN PDEVICE_OBJECT pdo)  
{  
    NTSTATUS status;  
    PDEVICE_OBJECT fdo;  
  
    // 在fdo中创建功能设备对象  
    status = IoCreateDevice(DriverObject,  
        sizeof(DEVICE_EXTENSION),  
        NULL,  
        FILE_DEVICE_UNKNOWN,
```



```

    0,
    FALSE,
    &fdo);
if( NT_ERROR(status))
    return status;
// 将fdo记录在设备扩展中
PWDM_DEVICE_EXTENSION dx =
    (PWDM_DEVICE_EXTENSION)fdo->DeviceExtension;
dx->pdo = pdo;
dx->fdo = fdo;
// 注册设备
status = IoRegisterDeviceInterface(pdo, &WDM_GUID, NULL,
    &dx->irSymLinkName);
if( NT_ERROR(status))
{
    IoDeleteDevice(fdo);
    return status;
}
IoSetDeviceInterfaceState(&dx->irSymLinkName, TRUE);
// 把设备对象放到设备堆栈上
dx->NextDevice = IoAttachDeviceToDeviceStack(fdo, pdo);
// 设置设备标志
fdo->Flags &= ~DO_DEVICE_INITIALIZING;
fdo->Flags |= DO_BUFFERED_IO;
// 注册WMI
// 调用WMI函数
return STATUS_SUCCESS;
}

```

需要说明的是上述代码最后调用RegisterWmi函数注册WMI需求，RegisterWmi函数的实现在Wmi.cpp中，9.4.4节将给出其源代码。

9.3.3节讨论了DispatchPnp分发例程应该实现的功能。在WdmDriver驱动程序中，只实现了最小的PnP支持，它能够响应AddDevice和IRP_MN_REMOVE_DEVICE调用，分别创建和清除FDO，其中对IRP_MN_REMOVE_DEVICE副功能代码的处理在DispatchPnp分发例程中。下面是实现代码：

```

NTSTATUS DispatchPnp ( IN PDEVICE_OBJECT fdo,
                     IN PIRP Irp)
{
    PWDM_DEVICE_EXTENSION dx=
        (PWDM_DEVICE_EXTENSION)fdo->DeviceExtension;
    // 获取副功能代码
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    ULONG MinorFunction = stack->MinorFunction;

```

```
// 将IRP传递给低层驱动程序
IoSkipCurrentIrpStackLocation(Irp);
NTSTATUS status = IoCallDriver(dx->NextDevice, Irp);

// 响应设备删除
if( MinorFunction==IRP_MN_REMOVE_DEVICE)
{
    IoSetDeviceInterfaceState(&dx->ifSymLinkName, FALSE);
    RtlFreeUnicodeString(&dx->ifSymLinkName);
    DeregisterWmi(fdo);
    if (dx->NextDevice)
        IoDetachDevice(dx->NextDevice);
    IoDeleteDevice(fdo);
}
return status;
}
```

需要说明的是，上述代码中灰色底纹处调用了DeregisterWmi函数，其功能为取消WMI功能注册，DeregisterWmi函数的实现在Wmi.cpp中，9.4.4节将给出其源代码。

电源管理功能在DispatchPower例程中实现。WdmDriver驱动程序的DispatchPower例程只是将IRP传递给低层驱动程序。代码如下：

```
NTSTATUS DispatchPower ( IN PDEVICE_OBJECT fdo,
                       IN PIRP Irp)
{
    PWDM_DEVICE_EXTENSION dx =
        (PWDM_DEVICE_EXTENSION)fdo->DeviceExtension;
    PoStartNextPowerIrp( Irp);
    IoSkipCurrentIrpStackLocation(Irp);
    return PoCallDriver( dx->NextDevice, Irp);
}
```

9.4.4 WMI支持

源文件Wmi.cpp实现对WMI的支持，该文件中包括注册WMI需求与撤消注册的两个函数、系统控制IRP的分发例程DispatchWmi以及由WMILIB用到的四个回调函数。下面分别介绍相应的源代码。

RegisterWmi函数实现注册WMI需求。该函数首先初始化WMILIB_CONTEXT结构的各个域，然后调用IoWMIRegistrationControl函数实现注册：

```
WMIGUIDREGINFO GuidList[1] =
{
    { &WMI_GUID, 1, 0 },
}
```

```
};

void RegisterWmi( IN PDEVICE_OBJECT fdo)
{
    PWDM_DEVICE_EXTENSION dx =
        (PWDM_DEVICE_EXTENSION)fdo->DeviceExtension;

    dx->WmiLibInfo.GuidCount = 1;
    dx->WmiLibInfo.GuidList = GuidList;

    dx->WmiLibInfo.QueryWmiRegInfo = QueryRegInfo;
    dx->WmiLibInfo.QueryWmiDataBlock = QueryDataBlock;
    dx->WmiLibInfo.SetWmiDataBlock = SetDataBlock;
    dx->WmiLibInfo.SetWmiDataItem = SetDataItem;
    dx->WmiLibInfo.ExecuteWmiMethod = NULL;
    dx->WmiLibInfo.WmiFunctionControl = NULL;

    IoWMIRegistrationControl(fdo, WMIREG_ACTION_REGISTER);
}

```

DeregisterWmi函数的功能与之正好相反，其代码如下：

```
void DeregisterWmi( IN PDEVICE_OBJECT fdo)
{
    IoWMIRegistrationControl(fdo, WMIRFG_ACTION_DEREGISTER);
}

```

DispatchWmi例程的主要代码已经在9.3.5节中给出，我们对其进行改造，使之适合**WdmDriver**驱动程序的需要。因为这些代码已经进行过解释，这里就不再重复了。

```
NTSTATUS WdmSystemControl(
    IN PDEVICE_OBJECT fdo,
    IN PIRP Irp)
{
    PWDM_DEVICE_EXTENSION dx =
        (PWDM_DEVICE_EXTENSION)fdo->DeviceExtension;
    NTSTATUS status;
    SYSCTL_IRP_DISPOSITION disposition;
    status = WmiSystemControl(&dx->WmiLibInfo, fdo, Irp, &disposition);
    switch(disposition)
    {
        case IrpProcessed:
            break;

        case IrpNotCompleted:
            IoCompleteRequest(Irp, IO_NO_INCREMENT);
    }
}

```

```
        break;

    default:
    case IrpForward:
    case IrpNotWmi:
        IoSkipCurrentIrpStackLocation(Irp);
        status = IoCallDriver(dx->NextDevice, Irp);
        break;
    }
    return status;
}
```

RegisterWmi函数设置了四个回调函数：SetDataItem、SetDataBlock、QueryDataBlock和QueryWmiRegInfo，分别处理9.3.5节中描述过的四种副功能代码的情况。其中SetDataItem和SetDataBlock两个函数只是简单地调用FailWMIRequest函数。下面是这些函数的源代码，请读者自己进行分析。

```
extern UCHAR SharedMemory[];

NTSTATUS FailWMIRequest(
    IN PDEVICE_OBJECT fdo,
    IN PIRP Irp,
    IN ULONG GuidIndex)
{
    PWDM_DEVICE_EXTENSION dx =
        (PWDM_DEVICE_EXTENSION)fdo->DeviceExtension;
    NTSTATUS status;
    if(GuidIndex==0)
        status = STATUS_INVALID_DEVICE_REQUEST;
    else
        status = STATUS_WMI_GUID_NOT_FOUND;
    status = WmiCompleteRequest(fdo, Irp, status, 0,
        IO_NO_INCREMENT);
    return status;
}

NTSTATUS SetDataItem(
    IN PDEVICE_OBJECT fdo, IN PIRP Irp,
    IN ULONG GuidIndex, IN ULONG InstanceIndex,
    IN ULONG DataItemId,
    IN ULONG BufferSize, IN PCHAR PBuffer)
{
    return FailRequest(fdo, Irp, GuidIndex);
}

NTSTATUS SetDataBlock(
```

```

    IN PDEVICE_OBJECT fdo, IN PIRP Irp,
    IN ULONG GuidIndex, IN ULONG InstanceIndex,
    IN ULONG BufferSize,
    IN PCHAR PBuffer)
{
    return FailRequest(fdo, Irp, GuidIndex);
}

NTSTATUS QueryDataBlock(
    IN PDEVICE_OBJECT fdo, IN PIRP Irp,
    IN ULONG GuidIndex,
    IN ULONG InstanceIndex,
    IN ULONG InstanceCount,
    IN OUT PULONG InstanceLengthArray,
    IN ULONG OutBufferSize,
    OUT PCHAR PBuffer)
{
    PWDM_DEVICE_EXTENSION dx =
        (PWDM_DEVICE_EXTENSION)fdo->DeviceExtension;
    NTSTATUS status;
    ULONG size = 0;
    switch (GuidIndex)
    {
    case 0:
    {
        ULONG SymLinkNameLen = dx->ifSymLinkName.Length;
        size = sizeof(ULONG)+sizeof(USHORT)+SymLinkNameLen;

        // Check output buffer size
        if (OutBufferSize<size)
        {
            status = STATUS_BUFFER_TOO_SMALL;
            break;
        }

        // Store uint32 BufferFirstWord
        *(ULONG *)PBuffer = *(ULONG*)SharedMemory;
        (UCHAR *)PBuffer += sizeof(ULONG);

        // Store string SymbolicLinkName as counted Unicode
        *(USHORT *)PBuffer = (USHORT)SymLinkNameLen;
        (UCHAR *)PBuffer += sizeof(USHORT);
        RtlCopyMemory(PBuffer, dx->ifSymLinkName.Buffer,

```



```
        SymLinkNameLen);

    // Store total size
    *InstanceLengthArray = size;

    status = STATUS_SUCCESS;

    break;
}
default:
    status = STATUS_WMI_GUID_NOT_FOUND;
    break;
}

status = WmiCompleteRequest( fdo, Irp, status, size,
    IO_NO_INCREMENT);

return status;
}

NTSTATUS QueryRegInfo(
    IN PDEVICE_OBJECT fdo, OUT PULONG PRegFlags,
    OUT PUNICODE_STRING PInstanceName,
    OUT PUNICODE_STRING *PRegistryPath,
    OUT PUNICODE_STRING MofResourceName,
    OUT PDEVICE_OBJECT *Pdo)
{
    PWDM_DEVICE_EXTENSION dx =
        (PWDM_DEVICE_EXTENSION)fdo->DeviceExtension;

    *PRegFlags = WMIREG_FLAG_INSTANCE_PDO;
    *PRegistryPath = &Servkey;
    RtlInitUnicodeString(MofResourceName, L"MofResource");
    *Pdo = dx->pdo;

    return STATUS_SUCCESS;
}
```

在9.3.5节中已经给出了MOF文件的内容，这里不再重复。

9.4.5 分发例程

WdmDriver驱动程序的分发例程在源文件Dispatch.cpp中，该文件定义了一个可供应用程序读写的4字节的共享内存缓冲区，并实现了创建、关闭、设备控制、读和写五个分发例程。

共享内存缓冲区定义如下：

```
const ULONG SHARED_MEMORY_SIZE = 4;
UCHAR SharedMemory[SHARED_MEMORY_SIZE];
```

WdmDriver驱动程序的创建、关闭和设备控制例程只是将IRP固定部分IoStatus域设置成指定的参数，并调用IoCompleteRequest，也就是说它们除了成功完成IRP外，其他什么也不做。下面是这些例程的源代码：

```
NTSTATUS DispatchCreate( IN PDEVICE_OBJECT fdo,
                      IN PIRP Irp)
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

NTSTATUS DispatchClose( IN PDEVICE_OBJECT fdo,
                     IN PIRP Irp)
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

NTSTATUS DispatchDeviceControl( IN PDEVICE_OBJECT fdo,
                              IN PIRP Irp)
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}
```

在写分发例程DispatchWrite中，首先得到当前I/O堆栈单元指针，并取得要传输的字节数。如果要传输的字节数大于共享内存缓冲区的大小，则返回STATUS_INVALID_PARAMETER，否则调用RtlMoveMemory函数从用户缓冲区将数据复制到共享内存缓冲区。最后，DispatchWrite例程将IRP固定部分的IoStatus域设置成指定的参数，并调用IoCompleteRequest函数成功完成IRP。下面是DispatchWrite例程的代码：

```
NTSTATUS DispatchWrite( IN PDEVICE_OBJECT fdo,
                     IN PIRP Irp)
{
```

```
PWDM_DEVICE_EXTENSION dx =
    (PWDM_DEVICE_EXTENSION) fdo->DeviceExtension;
PIO_STACK_LOCATION pIrpStack =
    IoGetCurrentIrpStackLocation(Irp);
NTSTATUS status = STATUS_SUCCESS;
ULONG BytesTxd = 0;

ULONG WriteLen = pIrpStack->Parameters.Write.Length;
if( WriteLen>SHARED_MEMORY_SIZE)
    status = STATUS_INVALID_PARAMETER;
else if( WriteLen>0)
{
    RtlMoveMemory( SharedMemory,
        Irp->AssociatedIrp.SystemBuffer, WriteLen);
    BytesTxd = WriteLen;
}

Irp->IoStatus.Status = status;
Irp->IoStatus.Information = BytesTxd;
IoCompleteRequest(Irp, IO_NO_INCREMENT);
return status;
}
```

读分发例程DispatchRead与写分发例程基本类似，下面是DispatchRead的源代码：

```
NTSTATUS WdmRead( IN PDEVICE_OBJECT fdo,
                IN PIRP Irp)
{
    PWDM_DEVICE_EXTENSION dx =
        (PWDM_DEVICE_EXTENSION) fdo->DeviceExtension;
    PIO_STACK_LOCATION pIrpStack =
        IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS status = STATUS_SUCCESS;
    ULONG BytesTxd = 0;

    ULONG ReadLen = pIrpStack->Parameters.Read.Length;

    if( ReadLen>SHARED_MEMORY_SIZE)
        status = STATUS_INVALID_PARAMETER;
    else if( ReadLen>0)
    {
        RtlMoveMemory( Irp->AssociatedIrp.SystemBuffer,
            SharedMemory, ReadLen);
        BytesTxd = ReadLen;
    }

    Irp->IoStatus.Status = status;
```

```

    Irp->IoStatus.Information = BytesTxd;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}

```

9.4.6 驱动程序的编译链接

上面已经完整地介绍了一个简单的WDM驱动程序WdmDriver的源程序，下面我们来把它编译链接成一个可以工作的驱动程序。

Windows 2000/XP下编写驱动程序的环境被称为DDK For Microsoft Windows 2000，或Windows 2000 DDK，其中DDK是Device Driver Kit（设备驱动程序）的缩写。Windows 2000 DDK可以从<http://www.microsoft.com/ddk>免费下载。

Windows 2000 DDK是一个命令行下的工作环境，在安装DDK之前需要安装Microsoft Visual C++和Win32 SDK（可选）。

编译链接器为Build.exe，它从配置文件Sources中读出待编译的程序的配置，包括源文件、目标文件等，从环境变量Include中得到引用文件的地址，然后调用Visual C++的编译链接器Nmake.exe进行实际的编译链接工作。日志文件build.log、build.wrn和build.err中分别记录了编译链接中执行的命令行、遇到的错误和遇到的警告。编译完成后的文件后缀为.sys。

有关Windows 2000 DDK的使用细节请参考DDK的相关文档。此外，在DDK的src目录下有许多示范程序可供参考。

Windows 2000 DDK 提供了一个称为WinDbg的调试工具，它可以让你在两台装有Windows 2000的计算机上进行源代码级调试。

由于驱动程序的结构比较复杂，所以在编写一个较复杂的驱动程序的过程中应分步来进行测试，在完成任何一部分工作后都应进行测试，以便及早地发现错误。

9.4.7 驱动程序的安装

Windows 2000/XP使用一个扩展名为INF的文本文件来控制与安装驱动程序相关的大部分活动。INF文件应该由驱动程序开发人员随驱动程序一起提供。通过INF文件可以告诉操作系统哪一个文件需要复制到用户硬盘上，应该添加或修改哪一个注册表项，如此等等。

WdmDriver驱动程序的INF文件内容如下：

```

[Version]
Signature="$Chicago$"
Class=Unknown
Provider=%THU%

[Manufacturer]
%THU% = WdmDriver

[WdmDriver]
%WdmDriver%=WdmDriver.Install, *WdmDriver

```

```
[DestinationDirs]

WdmDriver.Files.Driver=10,System32\Drivers
WdmDriver.Files.Driver.NT=10,System32\Drivers


[SourceDisksNames]
1="WdmDriver directory",,,,obj\i386\


[SourceDisksFiles]
WdmDriver.sys=1


;;;;;;;;;;;;;;
; Windows 98


[WdmDriver.Install]
CopyFiles=WdmDriver.Files.Driver
AddReg=WdmDriver.AddReg


[WdmDriver.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,WdmDriver.sys


[WdmDriver.Files.Driver]
WdmDriver.sys


;;;;;;;;;;;;;;
; Windows 2000


[WdmDriver.Install.NT]
CopyFiles=WdmDriver.Files.Driver.NT


[WdmDriver.Files.Driver.NT]
WdmDriver.sys,,,2


[WdmDriver.Install.NT.Services]
AddService = WdmDriver, 2, WdmDriver.Service


[WdmDriver.Service]
DisplayName      = "THU WdmDriver Example Driver"
ServiceType     = 1
StartType       = 3
ErrorControl    = 1
ServiceBinary   = %10%\System32\Drivers\WdmDriver.sys


;;;;;;;;;;;;;;;;
; Strings
```

```
[Strings]
THU="Tsinghua University"
WdmDriver="WdmDriver Example"
```

INF文件包含一些名字由方括号括起来的段，大部分段都含有一系列keyword = value形式的指令。

INF文件开始于一个Version段，该段确定文件中描述的设备类型。上述文件中，Signature必须是三种特定值之一：\$Chicago\$、\$Windows NT\$和\$Windows 95\$；Class确定设备类，Unknown表示未知的设备，Display表示显示适配器，FDC表示软盘控制器等。

接下来是Manufacturer段，其中列出了文件中有硬件描述的厂商。上述文件只列出了一个厂商%THU%=WdmDriver。每个厂商的硬件型号段（上述文件是WdmDriver段，对应Manufacturer段中的厂商描述）描述一个或多个设备。上述文件只描述了一个设备：%WdmDriver%=WdmDriver.Install, *WdmDriver，其中*WdmDriver标识一个硬件设备，WdmDriver.Install标识INF文件中的另一个段，该段含有为特定设备安装软件的指令。

接下来的三个段DestinationDirs、SourceDisksNames和SourceDisksFiles设置文件要复制到的目录名。另外，如果安装过程中需要多个磁盘，则需要知道哪个磁盘含有源文件。上述INF文件的这些段将使安装在源目录下的obj\i386\WdmDriver.sys文件复制到C:\Windows\System32\Drivers目录下。

接下来是Install段，该段包含指导安装程序安装所需驱动程序的实际指示。型号段WdmDriver指出Install段名，这个段名可以对应若干个加上与平台相关后缀的安装段。设备安装程序可以搜寻有最合适后缀的install段。如果安装到Windows 98中，它使用无后缀段；如果安装到Windows 2000中，将使用.NT段。

一个典型的Windows 2000安装段仅包含一个CopyFile指令，该指令指出我们想让安装程序用另一个INF段中（通常称为CopyFile段）的信息来指导文件复制。在WdmDriver驱动程序的安装文件中，CopyFile段是WdmDriver.Files.Driver.NT。

CopyFile段中的语句有下面通用形式：Destination, Source, Temporary, Flags。Destination是被复制文件出现在用户系统中的文件名；Source是源文件名，如果目标文件与源文件名字相同，则这里可以为空；如果安装的文件在安装过程中要被使用，可以在Temporary参数中指定一个临时名；Flags参数包含一个位掩码，用来对安装过程进行控制，例如是否要解压缩文件，是否允许跳过文件等。WdmDriver.Files.Driver.NT段的内容WdmDriver.sys,,,2指示安装程序把WdmDriver.sys复制到用户的硬盘上，Flags参数为2表示不允许跳过文件。

到现在为止所讨论的INF语法已经足够把驱动程序复制到用户硬盘上。另外我们还需要为PnP管理器做些安排，以使它知道应该载入哪个文件。服务段可用于此目的，该段名字由install段名再加上Services组成。以上面的INF文件为例，AddService指令中的2指出WdmDriver将成为该设备的功能驱动程序，这个指令的结果是使注册表的HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services中出现一个WdmDriver键。接下来的WdmDriver.Service段为内核模式（ServiceType为1）驱动程序定义服务项，内核模式驱动程序由PnP管理器按需求自动装载

(StartType为3)。在装入时发生的错误应该被登记,但不要阻止系统启动(ErrorControl为1)。执行映像为\Winnt\System32\Drivers\WdmDriver.Service(ServiceBinary的值)。

对于Windows 98,添加注册表项的工作是AddReg段负责的。该段中指令的形式为:

```
reg_root,[subkey],[value_entry_name],[flags],[value]
```

其中reg_root取值通常为KHR,对于WdmDriver而言。在Windows 98中,KHR对应HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Class\Unknown\0000,0000将由合适的Unknown设备号替换。

INF文件的最后一般是Strings段。INF文件中的%THU%一类值将由Strings段中的字符串全称替换。

有了INF文件,驱动程序的安装工作是十分简单的。首先进入控制面板,选择“添加新硬件”,从硬件列表中选择“其他设备”并单击“从软盘安装”。进入“从软盘安装”对话框之后单击“浏览”,找到WdmDriver.inf文件所在的文件夹,单击“确定”。从接下来的对话框中选择列出的硬件,并单击“下一步”,系统将把驱动程序WdmDriver.sys复制到Windows系统的system32\drivers文件夹中。

安装之后,WdmDriver应该出现在设备管理器的“其他设备”类别中,选中该设备,单击“属性”,可以查看该设备的驱动程序信息,如图9-13所示。



图9-13 WdmDriver驱动程序属性

9.4.8 驱动程序的测试

Windows的I/O管理器把每一个设备上层都抽象成了文件,所以在Win32用户程序中只要通过

简单的文件操作API函数就可以实现与驱动程序中的某个设备通信：

- **CreateFile** 打开一个设备，准备进行数据传输，返回一个与设备相关的句柄
- **CloseHandle** 关闭一个由CreateFile打开的设备
- **ReadFile** 从设备读取数据
- **WriteFile** 向设备写数据
- **DeviceIoControl** 对设备进行一些自定义的操作，比如更改设置等

为了对WdmDriver进行测试，编写了TestWdm.cpp文件。下面是该文件的源代码，请读者自行分析。

```
#include <windows.h>
#include <stdio.h>
#include "setupapi.h"
#include "initguid.h"
#include "guid.h"

HANDLE GetDeviceViaInterface( GUID* pGuid, DWORD Instance);

int main(int argc, char* argv[])
{
    // Open device
    HANDLE hWdmDevice = GetDeviceViaInterface((LPGUID)&WDM_GUID,0);
    if( hWdmDevice==NULL)
    {
        printf("Could not find open WdmDriver device\n");
        return 1;
    }
    printf("Opened OK\n");

    // Read what's left in buffer
    DWORD TxdBytes;
    ULONG Rvalue = 0;
    if( !ReadFile( hWdmDevice, &Rvalue, 4, &TxdBytes, NULL))
        printf("Could not read value\n");
    else if( TxdBytes==4)
        printf("Read successfully read stored value of 0x%X\n",Rvalue);
    else
        printf("Wrong number of bytes read: %d\n",TxdBytes);

    // Write
    ULONG Wvalue = 0xabcd01;
    if( !WriteFile( hWdmDevice, &Wvalue, 4, &TxdBytes, NULL))
        printf("Could not write %X\n",Wvalue);
}
```

```
else if( TxdBytes==4)
    printf("Write succeeded\n");
else
    printf("Wrong number of bytes written: %d\n",TxdBytes);

// Read
Rvalue = 0;
if( !ReadFile( hWdmDevice, &Rvalue, 4, &TxdBytes, NULL))
    printf("Could not read value\n");
else if( TxdBytes==4)
{
    if( Rvalue==Wvalue)
        printf("Read succeeded\n");
    else
        printf("Read succeeded, but got wrong value: %X',Rvalue);
}
else
    printf("Wrong number of bytes read: %d\n",TxdBytes);

// Check duff write fails
if( !WriteFile( hWdmDevice, &Wvalue, 5, &TxdBytes, NULL))
    printf("Duff Write correctly failed with error %d\n",GetLastError());
else
    printf("Duff Write unexpectedly succeeded\n");

// Close device
CloseHandle(hWdmDevice);
return 0;
}

HANDLE GetDeviceViaInterface( GUID* pGuid, DWORD instance)
{
    // Get handle to relevant device information set
    HDEVINFO info = SetupDiGetClassDevs(pGuid, NULL, NULL,
        DIGCF_PRESENT | DIGCF_INTERFACEDevice);
    if(info==INVALID_HANDLE_VALUE)
    {
        printf("No HDEVINFO available for this GUID\n");
        return NULL;
    }

    // Get interface data for the requested instance
    SP_INTERFACE_DEVICE_DATA ifdata;
    ifdata.cbSize = sizeof(ifdata);
```

```

if(!SetupDiEnumDeviceInterfaces(info, NULL, pGuid, instance, &ifdata))
{
    printf("No SP_INTERFACE_DEVICE_DATA available for this GUID\n");
    SetupDiDestroyDeviceInfoList(info);
    return NULL;
}

// Get size of symbolic link name
DWORD ReqLen;
SetupDiGetDeviceInterfaceDetail(info, &ifdata, NULL, 0, &ReqLen, NULL);
PSP_INTERFACE_DEVICE_DETAIL_DATA ifDetail =
    (PSP_INTERFACE_DEVICE_DETAIL_DATA)(new char[ReqLen]);
if( ifDetail==NULL)
{
    SetupDiDestroyDeviceInfoList(info);
    return NULL;
}
// Get symbolic link name
ifDetail->cbSize = sizeof(SP_INTERFACE_DEVICE_DETAIL_DATA);
if( !SetupDiGetDeviceInterfaceDetail(info, &ifdata,
    ifDetail, ReqLen, NULL, NULL))
{
    SetupDiDestroyDeviceInfoList(info);
    delete ifDetail;
    return NULL;
}

printf('Symbolic link is %s\n',ifDetail->DevicePath);
// Open file
HANDLE rv = CreateFile( ifDetail->DevicePath,
    GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE,
    NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

delete ifDetail;
SetupDiDestroyDeviceInfoList(info);
return rv;
}

```

习题

- 9.1 结合第6章的学习，简述设备驱动程序与硬件设备和操作系统的关系。
- 9.2 WDM驱动程序的基本特点是什么？

- 9.3 简述驱动程序堆栈的建立过程。
- 9.4 在分层的驱动程序堆栈中IRP是如何处理的？
- 9.5 简述WDM驱动程序的基本结构。从编程的角度考虑，试列出WDM驱动程序的基本框架。
- 9.6 在WDM驱动程序中，如何处理PnP？
- 9.7 WDM驱动程序是如何支持WMI的？
- 9.8 安装文件（.INF文件）有什么作用？如果不使用安装文件，能否安装设备驱动程序？如何安装？



实 习



实 习

实习1：进程同步

实习要求

在Windows 2000环境下，创建一个包含 n 个线程的控制台进程。用这 n 个线程来表示 n 个读者或写者。每个线程按相应测试数据文件的要求，进行读写操作。请用信号量机制分别实现读者优先和写者优先的读者-写者问题。

读者-写者问题的读写操作限制：

- 1) 写-写互斥。
- 2) 读-写互斥。
- 3) 读-读允许。

读者优先的附加限制：如果一个读者申请进行读操作时已有另一读者正在进行读操作，则该读者可直接开始读操作。

写者优先的附加限制：如果一个读者申请进行读操作时已有另一写者在等待访问共享资源，则该读者必须等到没有写者处于等待状态后才能开始读操作。

运行结果显示要求：要求在每个线程创建、发出读写操作申请、开始读写操作和结束读写操作时分别显示一行提示信息，以确信所有处理都遵守相应的读写操作限制。

测试数据文件格式

测试数据文件包括 n 行测试数据，分别描述创建的 n 个线程是读者还是写者，以及读写操作的开始时间和持续时间。每行测试数据包括四个字段，各字段间用空格分隔。第一字段为一个正整数，表示线程序号。第二字段表示相应线程角色，R表示读者是，W表示写者。第三字段为一个正数，表示读写操作的开始时间。线程创建后，延迟相应时间（单位为秒）后发出对共享资源的读写申请。第四字段为一个正数，表示读写操作的持续时间。当线程读写申请成功后，开始对共享资源的读写操作，该操作持续相应时间后结束，并释放共享资源。

下面是一个测试数据文件的例子：

```
1 R 3 5
2 W 4 5
```

```

3 R 5 2
4 R 6 5
5 W 5.1 3

```

与实验相关的API介绍

在本实验中可能涉及的API有：

1. 线程控制

CreateThread完成线程创建，在调用进程的地址空间上创建一个线程，以执行指定的函数；它的返回值为所创建线程的句柄。

```

HANDLE CreateThread (
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD
    DWORD dwStackSize,           // initial stack size
    LPTHREAD_START_ROUTINE lpStartAddress,    // thread function
    LPVOID lpParameter,          // thread argument
    DWORD dwCreationFlags,       // creation option
    LPDWORD lpThreadId           // thread identifier
);

```

ExitThread用于结束当前线程：

```

VOID ExitThread (
    DWORD dwExitCode // exit code for this thread
);

```

Sleep可在指定的时间内挂起当前线程：

```

VOID Sleep (
    DWORD dwMilliseconds // sleep time
);

```

2. 信号量控制

CreateMutex创建一个互斥对象，返回对象句柄：

```

HANDLE CreateMutex (
    LPSECURITY_ATTRIBUTES lpMutexAttributes, // SD
    BOOL bInitialOwner,           // initial owner
    LPCTSTR lpName                // object name
);

```

OpenMutex打开并返回一个已存在的互斥对象句柄，用于后续访问：

```

HANDLE OpenMutex (
    DWORD dwDesiredAccess, // access
    BOOL bInheritHandle,   // inheritance option
    LPCTSTR lpName         // object name
);

```

ReleaseMutex释放对互斥对象的占用，使之成为可用；

```
BOOL ReleaseMutex(  
    HANDLE hMutex    // handle to mutex  
);
```

WaitForSingleObject可在指定的时间内等待指定对象为可用状态:

```
DWORD WaitForSingleObject(  
    HANDLE hHandle,    // handle to object  
    DWORD dwMilliseconds // time-out interval  
);
```

实习2: 内存管理

实习要求

在本次实验中, 需要从不同的侧面了解Windows 2000/XP的虚拟内存机制。在Windows 2000/XP操作系统中, 可以通过一些API操纵虚拟内存。主要需要了解以下几方面:

- Windows 2000/XP虚拟存储系统的组织。
- 如何控制虚拟内存空间?
- 如何编写内存追踪和显示工具?
- 详细了解与内存相关的API函数的使用。

Windows 2000/XP虚拟内存机制简介

内存管理是Windows 2000/XP执行体的一部分, 位于Ntoskrnl.exe文件中, 是整个操作系统的重要组成部分。

默认情况下, 32位Windows 2000/XP上每个用户进程可以占有2 GB的私有地址空间, 操作系统占有剩下的2 GB。Windows 2000/XP在x86体系结构上利用二级页表结构来实现虚拟地址向物理地址的转换。一个32位虚拟地址被解释为三个独立的分量——页目录索引、页表索引和字节索引——它们用于找出描述页面映射结构的索引。页面大小及页表项的宽度决定了页目录和页表索引的宽度。比如, 在x86系统中, 因为一页包含4096字节, 所以字节索引被确定为12位宽 ($2^{12} = 4096$)。

应用程序有三种使用内存方法:

- 以页为单位的虚拟内存分配方法, 适合于大型对象或结构数组。
- 内存映射文件方法, 适合于大型数据流文件以及多个进程之间的数据共享。
- 内存堆方法, 适合于大量的小型内存申请。

本次实验主要是针对第一种使用方式。应用程序通过API函数 VirtualAlloc和VirtualAllocEx等实现以页为单位的虚拟内存分配方法。首先保留地址空间, 然后向此地址空间提交物理页面, 也可以同时实现保留和提交。

保留地址空间是为线程将来使用保留一块虚拟地址。在已保留的区域中, 提交页面必须指出将物理存储器提交到何处以及提交多少。提交页面在访问时会转变为物理内存中的有效页面。

相关的API函数

可以通过GetSystemInfo、GlobalMemoryStatus和VirtualQuery来查询进程虚空间的状态。主要的信息来源如下：

VOID GetSystemInfo (LPSYSTEM_INFO lpSystemInfo)

结构SYSTEM_INFO定义如下：

```
typedef struct _SYSTEM_INFO {
    DWORD dwOemId;
    DWORD dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAddress;
    DWORD dwActiveProcessorMask;
    DWORD dwNumberOfProcessors;
    DWORD dwProcessorType;
    DWORD dwAllocationGranularity;
    DWORD dwReserved;
} SYSTEM_INFO, *LPSYSTEM_INFO;
```

1. 函数VOID GlobalMemoryStatus (LPMEMORYSTATUS lpBuffer)

数据结构MEMORYSTATUS定义如下：

```
typedef struct _MEMORYSTATUS {
    DWORD dwLength;
    DWORD dwMemoryLoad;
    DWORD dwTotalPhys;
    DWORD dwAvailPhys;
    DWORD dwTotalPageFile;
    DWORD dwAvailPageFile;
    DWORD dwTotalVirtual;
    DWORD dwAvailVirtual;
} MEMORYSTATUS, * LPMEMORYSTATUS;
```

2. 函数DWORD VirtualQuery (LPCVOID lpAddress, PMEMORY_BASIC_INFORMATION lpBuffer, DWORD dwLength)

主要数据结构MEMORY_BASIC_INFORMATION定义如下：

```
typedef struct _MEMORY_BASIC_INFORMATION {
    PVOID BaseAddress;
    PVOID AllocationBase;
    DWORD AllocationProtect;
    DWORD RegionSize;
    DWORD State;
    DWORD Protect;
    DWORD Type;
} MEMORY_BASIC_INFORMATION;
```

```
typedef MEMORY_BASIC_INFORMATION * PMEMORY_BASIC_INFORMATION;
```

还有一些函数，例如VirtualAlloc，VirtualAllocEx，VirtualFree和VirtualFreeEx等，用于虚拟内存的管理。详情请见Microsoft的Win32 API Reference Manual。

实习内容

使用这些API函数，编写一个包含两个线程的进程：一个线程用于模拟内存分配活动，另一个线程用于跟踪第一个线程的内存行为。模拟内存活动的线程可以从一个文件中读出要进行的内存操作。每个内存操作包含如下内容：

- 时间：开始执行的时间。
- 块数：分配内存的粒度。
- 操作：包括保留一个区域、提交一个区域、释放一个区域、回收一个区域以及锁与解锁一个区域；可以将这些操作编号，存放于文件中。
- 大小：指块的大小。
- 访问权限：共五种PAGE_READONLY、PAGE_READWRITE、PAGE_EXECUTE、PAGE_EXECUTE_READ和PAGE_EXECUTE_READWRITE。可以将这些权限编号，存放于文件中。

跟踪线程将页面大小、已使用的地址范围、物理内存总量以及虚拟内存总量等信息显示出来。

实习3：快速文件系统

基本知识介绍

众所周知，CPU是整个计算机系统中运算速度最快的部分，而外部设备是最慢的部分，它们之间存在着很大的差别。然而，CPU却时时刻刻可能要求访问外设。如果CPU的每次操作都必须等待外设完成，那么CPU宝贵的运行时间就会大大浪费。随着现代计算机技术的发展，大多数现代操作系统都对这个问题进行了处理。下面就介绍两种Windows 2000中解决这个不匹配问题的方法：高速缓存和异步传输。

1. 文件高速缓存

文件高速缓存是CPU访问外设的一个“中间设备”。说是设备，其实它不是真正物理上的“设备”，而是一种核心级内存映像机制。由于它被设置在内存中，因此速度非常快，可以部分解决CPU与硬盘速度差异的问题。文件系统的驱动程序通过调用“高速缓存管理程序”来使用文件高速缓存，然后高速缓存管理程序执行高速缓存的处理工作。

文件高速缓存的原理是：假设一个进程读了文件的第一个字节，它常常会按照顺序读第二个、第三个字节，直到读出所有的字节。利用这个原理可以进行“预取”，也就是说，在进程没有要求读磁盘之前就先把文件读出来并放到高速缓存中。这样，当进程请求访盘时，高速缓存可以快速地把已经取到内存中的文件内容直接送给进程使用。从而大大加速了访盘速度。另外，由于一个文件可能会被多次读入，因此可以在第一次读入后，将文件数据保存在高速缓存中。这样，下

次再读时就不必再从硬盘而可以从缓存中读取。利用LRU (Least Recently Used, 最近最少使用) 的原则, 可以将不常使用的文件从缓存中删除以节省高速缓存空间。

另外, 文件高速缓存还有一个“事后写”的机制。具体地讲, 如果一个进程要求写磁盘, 它首先把要写的内容交给高速缓存。而高速缓存并不马上把它写到磁盘上, 而是寻找CPU空闲的时间来进行写操作。这样, 要写磁盘的进程就可以不必等待磁盘写完毕以后再继续工作, 这也就节省了整个进程的执行时间。这里需要说明的是, 如果有另外一个进程要访问还没有被写入磁盘的文件时, 高速缓存管理程序可以使这个进程直接读高速缓存里面新的即将要写入的文件内容, 而不是磁盘上的旧内容, 从而保证了文件内容的一致性。

2. 异步传输

与文件高速缓存不同, 文件的异步传输是一种改变指令执行顺序的机制。在以往的操作系统中, 指令都是顺序执行的, 下一条指令必须在上一条指令执行完毕后才能执行。因此, 如果CPU遇到一条访盘指令, 那么它就必须等待缓慢的磁盘访问结束以后才能进行后续的工作。如果它后面的指令并不依赖于访盘操作时, 这个等待就显得很没有必要。Windows 2000中使用了一种异步传输的机制来解决这个问题。它通过设置打开文件时的一个标志位 (见后) 来使进程不等待读写文件操作而继续执行。当后续指令必须用到磁盘访问的结果数据时, 它再通过一条Wait指令进行等待。这样, 在访盘指令和等待指令之间的指令就可以与磁盘访问同时进行了, 从而大大加快了系统的整个速度。

相关的API函数

(这里只是简单说明, 具体函数应用请参见MSDN 2000)

CreateFile函数说明

在MSDN中, CreateFile说明如下:

```
HANDLE CreateFile (
    . . . . .
    DWORD dwFlagsAndAttributes ,      // file attributes
    . . . . .
);
```

dwFlagAndAttributes参数可能用到的几个值:

FILE_FLAG_NO_BUFFERING: 没有文件高速缓存

FLAG_FLAG_SEQUENTIAL_SCAN: 使用文件高速缓存

FILE_FLAG_OVERLAPPED: 使用异步传输

其中FILE_FLAG_NO_BUFFERING| FILE_FLAG_OVERLAPPED (两个标志的“或”) 是用在异步传输实验中的参数, 系统将提供尽可能最好的异步传输性能。

实习内容

设计一个函数:


```
int filter(char source,char *sink,int f, char *fArg)
```

其中:

source: 源文件, 即从哪个文件读。

sink: 目标文件, 即写到哪个文件。

f: 一个对文件的操作 (可以指定任何操作, 如: 对每个字节都加1的操作)。

fArg: f操作需要的参数 (如果需要的话)。

这个函数的作用是从source文件中读数据, 通过操作f后, 将结果写入sink文件中。整个过程需要三个部分完成:

1) 建立 N 个 ($0 < N < 6$) 个filter函数 (不同功能的filter函数), 并自己做一个testFile, 作为File 1, 经过 N 步, 把File 1写入到文件File N 中。记录整个过程的时间。

因为这个过程的时间由许多不确定的因素构成, 所以, 需要经过多次反复实验才能得到最终数据。把实验做10次, 记录每次的时间并求出最终平均时间。在实验报告中写明你所使用的CPU速度和内存的大小。

2) 使用CreateFile的FILE_FLAG_SEQUENTIAL_SCAN标志位建立文件, 这样系统会给你的文件访问加上文件高速缓存。用1)中同样的方法计算平均时间, 并与1)中结果做比较。

3) 使用FILE_FLAG_OVERLAPPED标志位, 并将ReadFile和WriteFile与wait函数一起使用, 这样系统会实现异步传输。你可以改变程序中语句的顺序, 并记录改变顺序对速度的影响。和1)、2)一样计算平均时间, 并与1)、2)的结果进行比较。

实习4: 软盘I/O

基本知识介绍

从最早的MS-DOS系统开始到现在的Windows 2000/XP系统, 软盘一直都是广泛应用的辅助存储设备。最早的PC使用5.25寸、360K的软盘, 现在的软盘一般都是3.5寸、1.44M的。

Windows 2000/XP的文件管理器可以包容多种不同的磁盘设备、文件系统和磁盘格式。Windows 2000/XP自己定义并且使用的文件系统是NTFS; 对于软盘, Windows 2000/XP可以使用MS-DOS格式。本实习着重于了解磁盘的物理组织, 以及如何通过用户态的程序直接读写磁盘上的信息。

MS-DOS格式是一种比较早的文件系统格式, 也叫FAT格式。最初的FAT设计只能管理最大为32 MB的磁盘, 关于FAT格式的详细信息可以参考相关资料。

相关的API函数

CreateFile函数用于打开一个设备, 例如软盘, 该设备可以以虚拟文件的方式访问, 例如, 如果要打开驱动器A中的软盘并且按字节流的方式访问其内容, 可以使用

```
CreateFile(  
    "\\\\.\\A:",
```

```

    GENERIC_READ,
    FILE_SHARE_READ,
    NULL,
    OPEN_ALWAYS,
    0,
    NULL
);

```

CreateFile返回一个句柄，可供其他函数如ReadFile和WriteFile等使用，且可以在执行读写操作前通过调用SetFilePointer函数确定读写的位置。DeviceIoControl函数可以用于获取软盘的基本信息。

以上函数的详细说明和使用示例请参阅MSDN。

实习内容

在本实习中，需要实现的主要是三个函数：一个用于判定逻辑驱动器A中磁盘的基本信息；另一个用于读取磁盘的扇区；还有一个用于把磁盘上得到的信息输出到标准输出流中。另外，需要一个用户态的程序来调用这些函数，以确保这些函数可以正确工作。

这些函数的原型如下：

```

Disk physicalDisk(char driveLetter);
Void sectorDump(Disk theDisk, int logicalSectorNumber);
BOOL sectorRead(
    Disk theDisk,
    Unsigned logicalSectorNumber,
    Char *buffer
);

```

函数physicalDisk用于初始化磁盘，为随后的操作做好准备，参数driveLetter代表磁盘驱动器，在其实现代码中使用CreateFile打开设备并获取磁盘的基本信息。

函数sectorRead从指定的磁盘中读取一个给定扇区（logicalSectorNumber）的信息到一个缓冲区（buffer）中。如果读取操作成功，该函数应该返回TRUE。

函数sectorDump调用函数sectorRead，然后把结果输出到stdout中。可以考虑根据扇区的不同对得到的信息进行格式化输出，例如逻辑扇区0、1、10和19分别对应于启动扇区、FAT表1、FAT表2和根目录。

实现步骤

整个实习需要一张MSDOS格式的软盘，并且上面已经存有一些文件。

整个实习可以按照如下次序进行：

- 1) 实现函数physicalDisk。
- 2) 实现函数sectorRead和sectorDump，可以得到任何给定扇区的16进制输出。
- 3) 使用函数sectorDump查看磁盘的内容。

4) 编写程序调用physicalDisk、sectorRead和segmentDump三个函数，并验证其正确性。

实习5：用WinSock实现网络通信

实习要求

请在属于两个不同进程的线程之间实现通话功能。假设这种通话是非对称的，一个线程作为发起者，另一个作为接受者。发起者要建立一个会话时，它向接受者发出一个创建虚链路的请求。（即发起者作为客户端，接收者作为服务器端）。每个进程都有一个控制台终端用于发送和接受消息，消息以符号>开头表示发出的消息，以<开头表示收到的消息。请使用WinSock 2.0实现这个程序。

这个系统应该通过两个不同的进程演示，比如进程在不同的窗口以及不同的机器时无需改动源代码。系统的服务器端应该可以选择端口号，客户端在运行时确定端口号以及服务器的地址或者网络名称。

实习环境

当使用WinSock包时，请确定链接了WinSock 2.0库和C运行库，这需把wsock 32.lib和libc.lib加入库列表。

实习内容

通过WinSock编程可以实现因特网上的数据传输。只使用本机的进程和线程实际上就可以编写和调试这样的程序了。一旦程序可以在本机运行（使用IP地址），一般只需修改网络名称就可以使程序在网络上运行。

实习中，只需在控制台终端环境下键入hostname命令就可以知道自己所使用的机器的主机名称。如果使用实验室的机器，主机名称可能会根据不同的会话而改变，可以在程序中通过调用gethostname函数来获取相应的机器名。这个函数的声明如下：

```
struct HOSTENT FAR * gethostname( const char FAR * name );
```

传入ASCII字符参数就可以获得指向HOSTENT结构的指针。

```
struct hostent{
    char FAR * h_name;
    char FAR * FAR * h_aliases;
    short h_addrtype;
    short h_length;
    char FAR * FAR * h_addr_list;
};
```

查看联机文档了解HOSTENT结构各字段的含义。

在填写sockaddr_in结构的字段时要注意一点。阅读以下的代码可能会有一定帮助。

```
LPHOSTENT host;
```

```

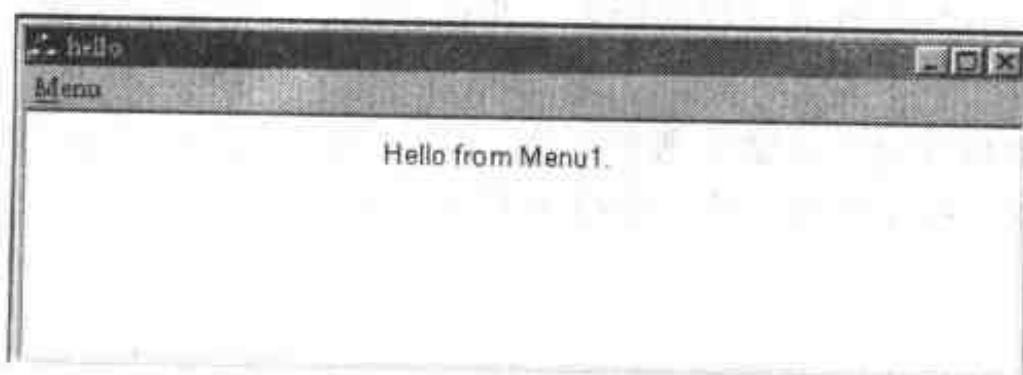
SOCKADDR_IN aServer;
...
host = gethostname(serverHostName);
ZeroMemory(&aServer, sizeof(SOCKADDR_IN));
aServer.sin_family = AF_INET;
aServer.sin_port = htons((u_short)port);
CopyMemory(&aServer.sin_addr, host->h_addr_list[0], host->h_length);

```

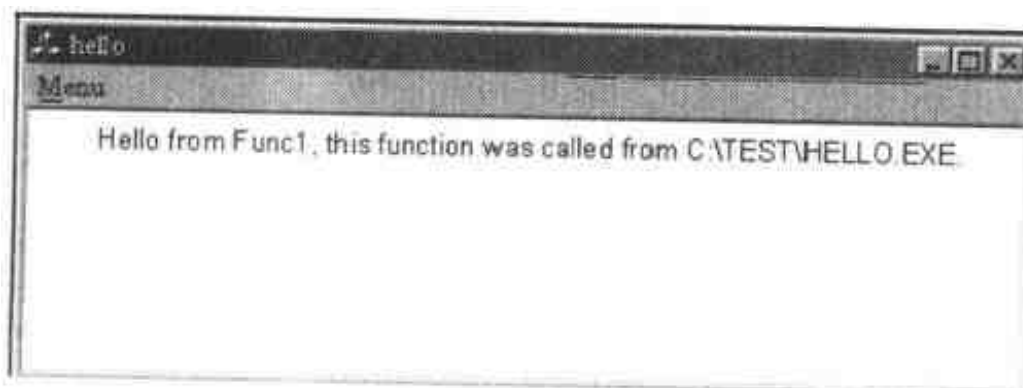
这个作业的难点之一就是编写的程序既要处理网络上传来的消息，又要处理来自控制台的标准输入，处理的先后应该是取决于哪一个事件先发生。这要求程序在socket和stdin上进行无阻塞的操作。这方面的内容可以查阅一些相关的联机文档。（提示：可以在控制台终端使用CreateFile函数。查阅MSDN的“Console and Port I/O”章节，其中对控制台I/O下一些例程的介绍可能是供一些有价值的思路。）

实习6: Windows应用程序与动态链接库

- (1) 编写一个Windows应用程序，要求产生下图所示的窗口。在该窗口的菜单栏中有一个Menu菜单，其中包含三个菜单项：Menu1、Menu2和Exit。单击菜单项Menu1，在窗口的客户区显示“Hello from menu1”。单击Menu2，在窗口的客户区显示“Hello from menu2”。单击Exit，退出程序。



- (2) 编写一个DLL，其中含有两个函数Func1和Func2可供应用程序调用，这两个函数的功能均为返回一个字符串。Func1返回的字符串是“Hello from Func1, this function was called from ...”，其中...为调用该函数的应用程序的名称与路径。Func2返回的字符串与此相似。
- (3) 修改第1项创建的应用程序，使得当单击菜单项Menu1时调用第2项创建的DLL中的Func1，获得Func1返回的字符串，并将其显示在窗口的客户区中，如下图所示。单击菜单项Menu2时调用Func2，并完成类似的操作。



要求使用C编程，不允许使用MFC，以便体会Windows应用程序的消息机制。关于GUI编程，请参考相关书籍。

实习7：WDM驱动程序开发

1. 构造WDM驱动程序开发环境。

首先安装Microsoft Visual C++。

从微软或其他站点下载Windows 2000 DDK，并将其安装到本机硬盘上。

2. 编辑、编译、连接并测试本章9.4节给出的WdmDriver例子。

在Microsoft Visual C++集成开发环境下编辑WdmDriver驱动程序的各个C语言源程序，并利用Microsoft Visual C++的编译器进行编译，发现并修改语法错误。

利用Windows 2000 DDK提供的BUILD实用程序对驱动程序进行编译，生成驱动程序的.SYS文件。有关BUILD实用程序的使用方法，请参考Windows 2000 DDK的相关文档。

按照9.4.7节的介绍编写WdmDriver驱动程序的INF文件，并安装上面用BUILD实用程序生成的驱动程序。

在Microsoft Visual C++集成开发环境下编辑WdmDriver驱动程序的测试程序，通过编译链接生成可执行程序。运行测试程序，观察运行结果。

说明：

1) WdmDriver驱动程序及其测试程序中有一些函数在本章正文中未做解释，请参考Windows 2000 DDK的相关文档。

2) 如果在编译驱动程序时提示缺少哪个文件，可以用Windows的搜索工具先找到这个文件，然后将此文件拷入编译驱动程序时所用的工作目录下再进行编译。

术

语

Access Violation 访问违规

Active/Valid 活动/有效

Address Space 地址空间

Address Translation 地址转换

Address Windowing Extension (AWE)

地址窗口扩充

Advanced Programmable Interrupt Controller

(APIC) 高级可编程中断控制器

Affinity Mask 亲和掩码

Allocation Granularity 分配粒度

Application Program Interface (API) 应用程序接口

Asymmetric Multiprocessing (ASMP) 非对称多处理器

Asynchronous I/O 异步I/O

Asynchronous Procedure Call (APC) 异步过程调用

Asynchronous Read-ahead With History 带历史信息的异步预读

Atomic Transaction 原子事务

Attribute List 属性列表

Balance Set Manager 平衡集管理器

Base Address 基址

Base File Record 基文件记录

Basic Disk 基本盘

Best-fit 最佳适配

Binary Semaphore 二进制信号量

Blocked 阻塞(状态)

Buffer 缓冲区

Built-in 内置

Bus Driver 总线驱动程序

C2 Security Requirement C2安全需求

Cache Coherency 高速缓存的一致性

Cache Management 高速缓存管理

Cache Manager 高速缓存管理器

CDROM File System (CDFS) 只读光盘文件系统

Checkpoint Record 检查点记录

Class/Port/Miniport 类/端口/小端口

Clock Algorithm 时钟算法

Collided Page Fault 缺页冲突

Common Internet File System (CIFS) 通用因特网文件系统

Compaction 内存紧缩

Condition 条件变量

Consumable Resource 可消耗资源

Control Object 控制对象

Copy-on-write 写时复制

Critical Section 临界区

Cryptographic Service Provider (CSP) 加密服务提供者

Daisy-chain 菊花链

Data Decryption Field (DDF) 数据解密字段

Data Encryption Standard (DES) 数据加密标准

Data Recovery Field (DRF) 恢复密钥字段

Deadlock 死锁

Debugger 调试器

Deferred Procedure Call (DPC) 延迟过程调用
Demand Paging 请求调页
Demand Zero 请求零页
Dereference Segment Thread 废弃段线程
Device Driver 设备驱动程序
Device IRQL 设备中断优先级
Device Object 设备对象
Direct Memory Access (DMA) 直接存储器存取
Dispatch Routine 调度(分发)例程
Dispatch 调度, 分发
Distributed Component Object Model(DCOM) 分布式组件对象模型
Domain Name System (DNS) 域名系统
Drive Letter 驱动器名
Driver Object 驱动程序对象
Driver Support Routine 驱动程序支持例程
Driver Verifier 驱动程序检查器
Dynamic Disk 动态盘
Dynamic Partitioning 动态分区
Dynamic-link Library 动态链接库
Encrypted File System (EFS) 加密文件系统
Entry Section 进入区
Environment Subsystem 环境子系统
Exception Dispatcher 异常调度程序
Exception 异常
Executive 执行体
Exit Code 退出代码
Exit Section 退出区
Extended Partition 扩展分区
Fast I/O 快速I/O
Fast LPC 快速LPC
Fault Tolerance 容错
File Allocation Chain 文件分配链

File Allocation Table (FAT) 文件分配表
File Control Block (FCB) 文件控制块
File Encryption Key (FEK) 文件密钥
File Mapping 文件映射
File Server 文件服务器
File System Driver (FSD) 文件系统驱动程序
First Come First Service 先来先服务(算法)
First-fit 首先适配
Foreground Quantum Boost 前台进程时间配额提升(字段)
Frame Locking 页框锁定
Handle 句柄
Handler Routine 处理程序例程
Hard Real-time System 硬实时系统
Hardware Abstraction Layer (HAL) 硬件抽象层
Heap 堆
High Performance File System (HPFS) 高性能文件系统
Highest Response Ratio Next 最高响应比优先
Hyperspace 超空间
I/O Request Packet (IRP) I/O请求包
Intelligent Read-ahead 智能预读
Interlock 互锁
Inter-Process Communication (IPC) 进程间通信
Inter-Processor Interrupt 处理器间中断
Interrupt Dispatch Table(IDT) 中断调度表
Interrupt Dispatcher 中断调度程序
Interrupt Object 中断对象
Interrupt ReQuest Level (IRQL) 中断请求优先级
Interrupt Request 中断源编号
Interrupt Service Routine (ISR) 中断服务例程

Invalid PTE 无效页表项	Logical Partition 逻辑分区
Kernel Mode Device Driver 核心态设备驱动程序	Logical Sequence Number (LSN) 逻辑序列号
Kernel Mode Graphics Driver 核心态图形驱动程序	Logon Process 登录进程
Kernel Mode 核心态	Mailslot 邮件槽
Kernel Object 内核对象	Mapped File 映射文件
Kernel 内核	Mapped Page Writer 映射页面写入器
Kernel's Dispatcher 内核调度器	Master File Table (MFT) 主控文件表
Kernel-level Thread 内核线程	Memory Management Unit (MMU) 内存管理单元
Key Ring 密钥链	Message Queue 消息队列
LANMan Redirector LANMan重定向器	Metadata 元数据
Lazy Writing 延迟写	Microsoft Distributed Transaction Coordinator (MS DTC) 微软分布式事务协调器
Least Frequently Used (LFU) 最不常用 (算法)	Microsoft Management Console (MMC) 微软管理控制台
Least Recently Used (LRU) 最近最久未使用 (算法)	Microsoft Transaction Server (MTS) 微软事务服务器
Legacy Application 遗留应用程序	Mirrored Volume 镜像卷
Lightweight Process 轻量级进程	Modified Page Writer 已修改页面写入器
Local Procedure Call (LPC) 本地过程调用	Monitor 管程
Local Security Authority Server (LSASRV) 本地安全授权服务器	Multicast Multimedia Conferencing 组播多媒体会议
Local Security Authority Subsystem (LSASS) 本地安全授权子系统	Multipartition Volume 多分区卷
Local Security Authority (LSA) Server Policy Database 本地安全授权服务器策略数据库	Multiple Provider Router (MPR) 多提供者路由器
Local Security Authority (LSA) Server 本地安全授权服务器	Multiple UNC Provider (MUP) 多UNC提供者
Lock Bit 锁定标志位	Multiple-level Queue 多级队列 (算法)
Log File Service (LFS) 日志文件服务	Mutual Exclusion 互斥
Log File 日志文件	Named Pipe 命名管道
Logical Block 磁盘逻辑块	Namespace Service Provider 名字空间服务提供者
Logical Cluster Number (LCN) 逻辑簇号	Namespace 名字空间
Logical Disk Manager (LDM) 逻辑磁盘管理器	NetBIOS Emulator NetBIOS仿真器
	NetBIOS Frame Format NetBIOS帧格式

Network And Dial-Up Connection Application
网络和拨号连接应用程序

Network Basic Input / Output System 网络
基本输入输出系统

Network Driver Interface Specification(NDIS)
网络驱动程序接口规范

Network-Resource Name Resolutin 网络资
源名字解析

Next-fit 下次适配

Nonpaged Pool 非分页缓冲池

Nonresident Attribute 非常驻属性

Not Recently Used (NRU) 最近未使用
(算法)

Numeric Processor Extension 协处理器扩展

Object Attribute 对象属性

Object Directory 对象目录

Object Handle 对象句柄

Object Manager Namespace 对象管理器
名字空间

Object Manager 对象管理器

Object Type 对象类型

Opportunistic Lock 机会锁

Optimal 最佳(算法)

Overlay 覆盖

Page 页(面)

Page Buffering 页面缓冲(算法)

Page Directory Entry (PDE) 页目录项

Page Directory 页目录

Page Fault 缺页

Page Fault Handling 缺页处理

Page File 页文件

Page Frame 页框

Page Frame Number (PFN) 页框号

Page Table Entry (PTE) 页表项目

Page Table 页表

Paged Pool 分页缓冲池

Partition 分区

Passive Level 被动中断优先级

Performance Option 性能选项

Physical Address Extension (PAE) 物理
地址扩展

Pipe 管道

Port Driver 端口驱动程序

Power Fail Level 电源故障中断优先级

Prepaging 预调页

Primary Partition 主分区

Principle of Locality 局部性原理

Priority Scheduling 优先级调度(算法)

Private Semaphore 私有信号量

Process Control Block (PCB) 进程控制块

Process Group 进程组

Process ID 进程标识符

Process/Stack Swapper 进程/堆栈交换程序

Process's Priority Class 进程优先级类

Processor Control Block 处理器控制块

Processor Control Region 处理器控制区

Processor Status Word 处理器状态字

Profile Level 配置文件级

Programmable Interrupt Controller 可编程
中断控制器

Prototype PTE 原型页表项

Quantum Unit 时间配额单位

Quantum 时间配额

RAID 廉价冗余磁盘阵列

Reader-Writer Problem 读者-写者问题

Real-Time Collaboration (RTC) 实时协作

Real-Time Level 实时(线程)优先级

Recoverable File System 可恢复的文件系统

Recovery Agent 恢复代理

Redirector File System Driver (Redirector
FSD) 重定向器FSD

Remainder Section 剩余区

- Reparse Point 再解析点
 Replacement Policy 置换策略
 Resident Attribute 常驻属性
 Resource Allocation Graph 资源分配图
 Reusable Resource 可重用资源
 Rollback 回退
 Roll-in/Roll-out 滚进/滚出
 Round Robin with Multiple Feedback 多级反馈队列 (算法)
 Round Robin 时间片时钟 (算法)
 RSA (Rivest Shamir Adleman) 由 Rivest-Shamir-Adleman 发明的一种提供秘密信息和数字签名的密文系统
 Scatter/Gather I/O 分散/集中 I/O
 Scheduling 调度
 Section Object 区域对象
 Sector Sparing 扇区备用
 Secure Attention Sequence (SAS) 安全注意序列
 Security Credential 安全证书
 Security Descriptor 安全描述符
 Security ID (SID) 用户安全标识
 Security Reference Monitor (SRM) 安全引用监视器
 Segment 段
 Self-balancing Binary Tree 自平衡二叉树
 Semaphore 信号量
 Server Message Block (SMB) 服务器消息块 (协议)
 Server Process 服务器进程
 Service Control Manager (SCM) 服务控制管理器
 Session Space 会话空间
 Shared Memory 共享存储区
 Shortest Job First (最)短作业 (优先算法)
 Shortest Process Next (最)短进程 (优先算法)
 Shortest Remaining Time 最短剩余时间 (优先算法)
 Signal 信号
 Simple Volume 简单卷
 Slot 槽
 Socket 套接字
 Soft Real-Time System 软实时系统
 Spanned Volume 跨分区卷
 Sparse Multilevel Index Array 多维索引的稀疏数组
 Spinlock 自旋锁
 Stand By 后备
 Standard Network Resource Naming Convention 标准网络资源命名规范
 Starvation 饥饿
 Stream-based Caching 基于流的高速缓存
 Striped Volume 条带卷
 Structured Exception Handling 结构化异常处理
 Subsystem Dynamic-link Library 子系统动态链接库
 Suspend 挂起 (状态)
 Swapping 交换
 Symbolic Link 符号链接
 Symmetric Encryption Algorithm 对称加密算法
 Symmetric Multiprocessing (SMP) 对称多处理
 Synchronization 同步
 Synchronous I/O 同步 I/O
 System Cache 系统高速缓存
 System Control Block (SCB) 系统控制块
 System Support Processes 系统支持进程
 System Thread 系统线程

Task Manager 任务管理器	Virtue Address Control Block (VACB)
Task 任务	虚拟地址控制块
TDI Transport TDI传输口	Virtue Address Descriptor (VAD) 虚拟地址描述符
Telephony API (TAPI) 电话API	Volume Namespace 卷名空间
Telephony 电话服务	Volume Parameter Block (VPB) 卷参数块
Terminal Service 终端服务	Win32 Application Programming Interface (API) Win32应用程序编程接口
Thread Scheduling 线程调度	Windows 2000 Professional Windows 2000专业版
Threshold 阈值	Windows 2000 Server Windows 2000服务器版
Transition 转换	Windows 2000/XP Internal Routine Windows 2000/XP内部例程
Translation Look-aside Buffer (TLB) 关联存储器, 快表, 后备表	Windows 2000/XP Service Windows 2000/XP服务
Transport Protocol 传输协议	Windows 2000/XP System Service Windows 2000/XP系统服务
Transport Provider Interface 传输提供者接口	Windows Internet Name Service (WINS) Windows因特网命名服务
Transport Service Provider 传输服务提供者	Windows Networking API 微软网络 API
Trap Dispatcher 陷阱调度程序	Winsock Windows套接字
Trap Frame 陷阱帧	Working Set Manager 工作集管理器
Trap Handler 陷阱处理程序	Working Set 工作集
Trap 陷阱	Worst-fit 最坏适配
Type Object 类型对象	Write Throttling 写阻塞
Unallocated Hole 未分配空洞	Write-ahead Logging 预写日志
Universal Disk Format (UDF) 通用磁盘格式	Write-back 回写
Update Record 更新记录	Write-through 通写
User ID 用户标识符	Zero Copy Send 零拷贝发送
User Mode 用户态	Zero Page Thread 零页线程
User-level Thread 用户线程	Zeroed 零初始化
Variable Level 可变(线程)优先级	
View 视图	
Virtual Address Read-ahead 虚拟地址预读	
Virtual Block Caching 虚拟块缓存	
Virtual Cluster Number (VCN) 虚拟簇号	
Virtual Memory 虚拟存储器	

参考文献

1. David A. Solomon Mark E. Russinovich. Inside Windows 2000. Microsoft Press, 2000
2. Gary Nutt. Operating System Projects Using Windows NT. Addison-Wesley, 1999
3. Gary Nutt. Operating Systems: A Modern Perspective. Addison-Wesley, 1997
4. Andrew S. Tanenbaum. Modern Operating Systems. Prentice Hall, 1998
5. Abraham Silberschatz and Peter Baer Galvin. Operating System Concepts. Addison-Wesley, 1998
6. Walter Oney. Programming the Microsoft Windows Driver Model. Microsoft Press, 2000
7. Art Baker. The Windows NT Device Driver Book: A Guide for Programmers. Prentice Hall PTR, 1979